Delft University of Technology

BSc Electrical Engineering

Automatic Direction Finding Techniques for Precise Tracking of Lightweight Transmitters

BSc Thesis

Authors:
Noura de Klerk
Yue Kroeze

Date:
June 28, 2024

Supervisor:
dr. Tomás Manzaneque

1 Abstract

This thesis investigates the usage of automatic direction finding in the Asian Hornet tracking field. A lightweight and portable design with a precision of 0.5 degrees is discussed and implemented. The goal of this localization system is to track the transmitter that is attached to the Asian Hornet. The system utilizes two Software Defined Radio (SDR) dongles from the brand RTL that receive the transmitted signal from the transmitter and send it to the computer. The Angle Of Arrival of the signal is calculated using phase interferometry. The results show that the Python program, the algorithm implementation in combination with the used hardware, used to calculate the phase difference is not fast enough to process the received signal in real time. Furthermore, the chosen RTL SDR dongles cannot be synchronized easily which leads to unknown and varying phase differences when receiving the signals.

2 Preface

The context of this thesis is the Bachelor Graduation Project of the Electrical Engineering Bachelor given by the Technical University of Delft. Therefore this report was written by two third year Electrical Engineering students. This project was proposed by beekeeper Henk Mezger.

While this report provides a clear explanation of the concepts, it is intended for readers who already possess a basic understanding of both telecommunications and signal processing. This foundational knowledge will enable them to fully grasp the material presented.

Our gratitude goes out to our supervisor dr. Tomás Manzaneque and our technical advisor Ing. Jeroen Bastemeijer. Furthermore, we want to thank our group members Cem Çetiner, Yunus Emre Döngel, Emmad Hassan and Sofya Mikhaylitskaya for the pleasant collaboration. Lastly we want to thank prof. dr. ir. Alle-Jan van der Veen for answering all our questions regarding signal processing and António Maia Barros for assisting us with the prototyping.

Delft, June 28, 2024
Noura de Klerk and Yue Kroeze

Contents

1 Abstract **i**

2 Preface **ii**

3 Introduction **2**

4 Theory **3**
4.1 Hornet specifications . . . 3
4.2 Software defined radio . . . 3
4.3 Automatic direction finding methods . . . 4
4.4 Localizing the transmitter . . . 6
4.5 Challenges for direction finding . . . 8
4.6 Precision estimation . . . 11
4.7 Automatic direction finding versus manual direction finding . . . 12
4.8 User interface . . . 13

5 Program of requirements **14**
5.1 External requirements . . . 14
5.2 Stakeholders . . . 14
5.3 User stories . . . 14
5.4 Requirements . . . 15

6 Methodology **16**
6.1 Circuit of the receiver . . . 16
6.2 Explanation of the algorithm . . . 18
6.3 Development of the system . . . 19
6.4 Receive two signals . . . 22
6.5 Phase difference calculation with the received signal . . . 24
6.6 Phase difference between two antennas . . . 24
6.7 Angle of arrival calculation . . . 26
6.8 Received signal strength indication and distance measurement . . . 26
6.9 Direct phase difference . . . 27
6.10 Gold code improvement . . . 28

7 Analysis of the system **29**
7.1 Algorithm and Python drawbacks . . . 29
7.2 Synchronization of the dongles . . . 29
7.3 Precision estimation of the algorithm . . . 29
7.4 Received signal strength indication . . . 30
7.5 Testing the user interface . . . 30

8 Discussion **31**

9 Conclusion **31**

10 Future work **31**

11 Bibliography **32**

A Additional figures **34**

B Application code **36**

C Test code **56**

D Calculations **76**

3 Introduction

The Asian Hornet (Vespa velutina) is an exotic animal in the Netherlands which presence endangers the existence of the honeybees[1]. To shrink down the Asian Hornet population, the nests are destroyed during the end of the season. In order to find these nests, Asian Hornet workers are captured and tagged with a transmitter which is then followed to the nest. Current solutions for the receiver are dependent on the operator to find the signal manually and follow it. This report suggests an automatic solution which uses automatic direction finding localization. Beekeepers have been trying to fight off the enlargement of the population. This is without success however as the population has been growing for years. With the use of automatic direction finding, the tracking must become more efficient, leading to more nests being found on a monthly basis. Figure 1 show the idea in action. A transmitter is placed on the hornet and an operator uses a receiver with direction finding to know where to go. The objective of this thesis is to make a system which can tell the direction of an incoming signal.

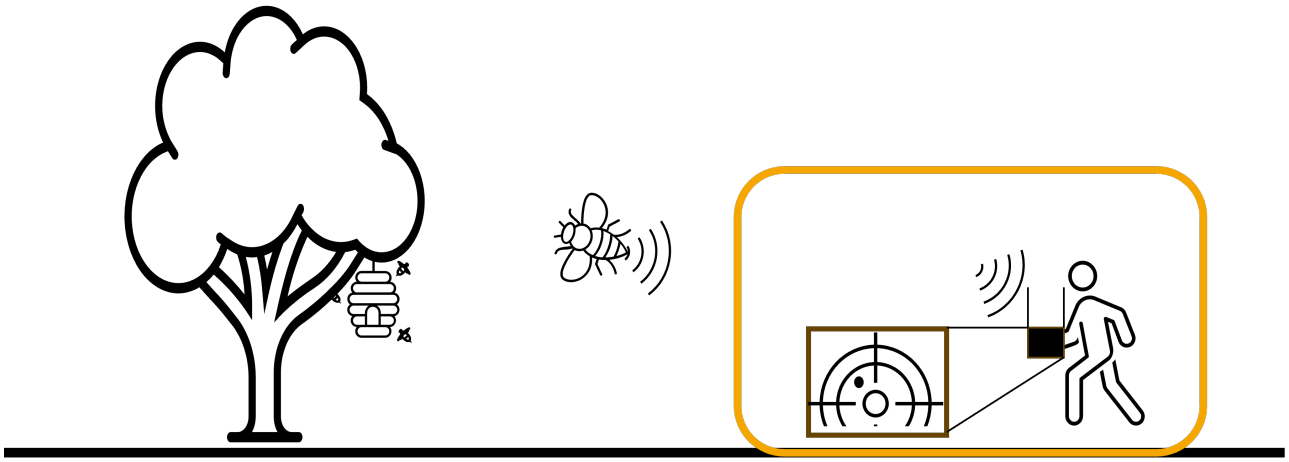


Figure 1: Overview of the tracking process with automatic direction finding

Currently, there are two other companies on the market that sell transmitters for the tracking of the Asian Hornet: Lowlands[2] and Lotek[3]. The Lowlands tracker works at a frequency of 150MHz and sends a signal regular intervals. The Lotek counterpart uses a frequency between 138 MHz and 174 Mhz. These transmitters are used with a receiver connected to a directional antenna in such a way that the system can be used with manual direction finding by turning the antenna and looking for a higher signal strength.

This report describes a receiver system that uses two antenna's to find the Angle Of Arrival of an incoming signal. Section 4 will go into the background information that is essential for the system. The requirements are then discussed in Section 5. Section 6 gives an elaborate overview of the process of building the system followed by an analysis of the product in Section 7. This report ends with a conclusion and discussion of the final system in Sections 9 and 8.

4 Theory

4.1 Hornet specifications

The most important characteristic of the Asian Hornet for this project is their flight capability. This includes flight speed and the distance they are willing to get away from their nests. While its European counterpart can only fly 1.86 m/s, the Asian hornet can reach speeds up to 6 m/s [1]. As for the foraging range, different values have been found with an average of 700 m with an observed maximum of 2 km, [1]. However, these characteristics can change heavily with certain factors, one of them being weather.

4.2 Software defined radio

Software defined radio (SDR) is a radio communication system where components that have been typically implemented in hardware are instead implemented with the use of software. These components are for instance, mixers, amplifiers, filters et cetera. There are three required hardware components for this. The first being a SDR receiver (this is also called a dongle), the second is one or multiple antenna's and the third being a computer running SDR software [4], [5]. The SDR first captures the signal using the antenna. Then the frequency is down-converted to be centered around 0 Hz. As opposed to analog devices, not one but two mixers are used. A mixer can down-convert or up-convert signals using a local oscillator. The formula of this can be seen in Equations 1 and 2. To not discard the data that is now down-converted to negative frequencies, a 90 degree shift is used to convert them into imaginary frequencies. Next, the signal is filtered to lower the noise. A schematic of this can be seen in Figure 2 [6].

$$f1 = f_{signal} + f_{lo} \tag{1}$$

$$f2 = f_{signal} - f_{lo} \tag{2}$$

4.2.1 IQ data

IQ sampling, or complex sampling, means that there are two orthogonal waves. One in phase(I) and one quadrature (Q) wave shifted 90°. This way, an array of complex numbers, one sample being represented by $I + jQ$, can show the phase and amplitude of the received signal, as can be seen in Equation 3.

$$A \cdot cos(2\pi ft + \phi) = (I^2 + Q^2)^{1/2} \cdot cos(2\pi ft + tan^{-1}(\frac{Q}{I})) \tag{3}$$

An SDR uses direct conversion, or zero IF, to down convert the signal from its carrier frequency to base-band, 0 Hz, see Figure 2. This results in a complex signal that can be represented with IQ data, I being the real component and Q the imaginary component.

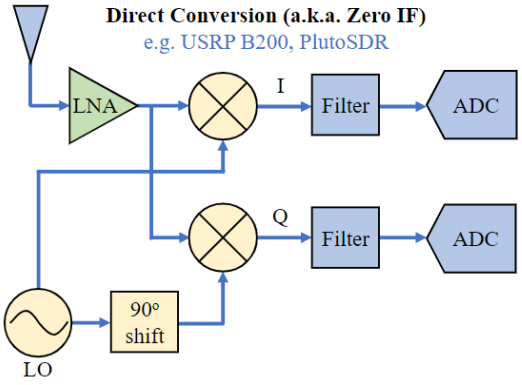


Figure 2: Schematic of direct conversion in an SDR. Image from [7]

4.3 Automatic direction finding methods

The receiver will act as a single bearing. Multiple bearings do make the localization more accurate, but given that the goal is to track the transmitter opposed to pin pointing its location, one bearing suffices. There are multiple direction finding methods. In the following section, three of these are considered for this project.

4.3.1 Pseudo-Doppler direction finding

Pseudo-Doppler direction finding is a frequency based direction finding and a relative simple way of finding the direction of the signal source. Doppler direction finding uses the Doppler effect [8]. This effect is shown in Figure 3 [9].

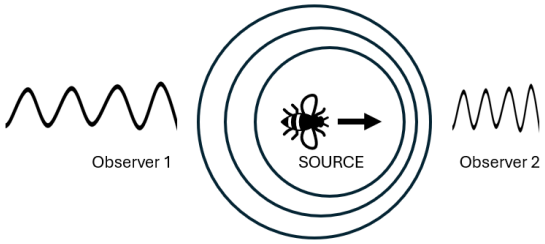


Figure 3: Visual explanation of Doppler shift. When the sound source is moving away from the observer, the frequency is lower then when the sound source is approaching.

This shows that when the receiver moves towards the transmitter, the receiver frequency moves upwards and vice versa. To use this Doppler shift, the vertical polarized antenna should be rotated in a circular fashion at a very high speed. Figure 4 shows how a rotating antenna makes a sinusoidal wave in the graph. These values are used to determine the direction.

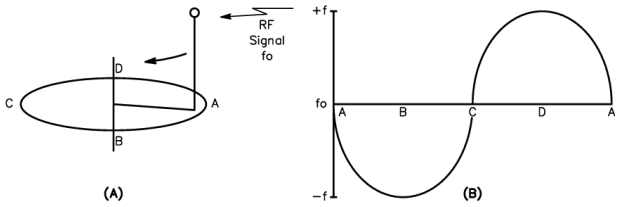


Figure 4: Visual explanation of Doppler shift using a rotating antenna. (A) shows the in incoming RF signal to the antenna. (B) shows how the sinusoidal wave that is obtained by measuring the frequency shift at the different positions of the rotating antenna. Image from [10]

Given that rotating a single antenna at a high speed is not practical, pseudo-Doppler is used in direction finding. Instead of moving a singular antenna, multiple antennas are used which are placed on the circumference of a circle. One of the antennas is turned on a any given time and its signal is read out. The antennas alternate in a circular fashion. With the use of an algorithm the angle of arrival of the incoming signal is then determined [11].

4.3.2 Watson-Watt direction finding

Watson-Watt is an amplitude based direction finding system. It compares the amplitudes of the received signal at each Adcock antenna. An Adcock antenna array consist of two antenna pairs which are placed in a cross manner. Each of the antenna pairs creates a figure eight with its range [12]. A visualisation of this is given in Figure 5. The Watson-Watt setup can be seen in Figure 6.

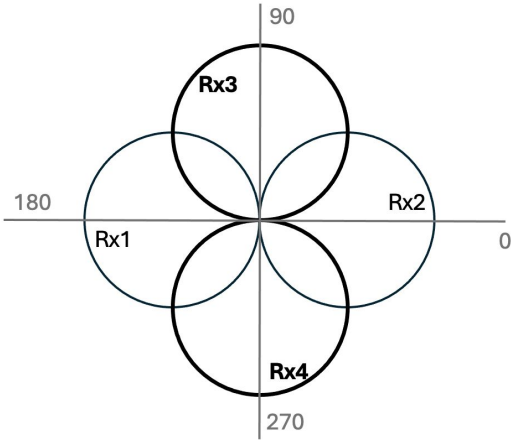


Figure 5: Range of the Adcock antennas.

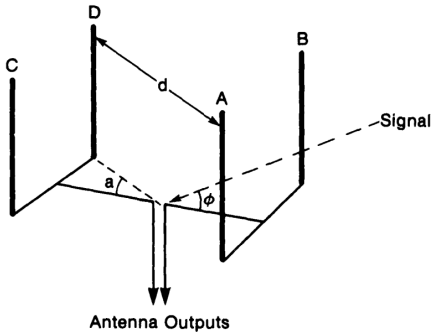


Figure 6: Schematic overview of the antenna array for the Watson-Watt direction finding method. Image from [12].

The sensitivity is at a maximum along the axis. This antenna pattern gives a unique set of magnitudes for every direction. Watson-Watt can not measure elevation. Furthermore, the azimuth accuracy decreases as the transmitter elevation increases or decreases.

4.3.3 Phase interferometry

With the use of two antennas, the phase shift of the incoming transmitted signal can be calculated. Using this phase shift (delta x), the angle of arrival of the transmitted signal is obtained [12]. The measurement setup for this method can be seen in Figure 7. In this setup the incoming signal is assumed to be in the far field of the antennas and is therefore treated as a plane wave.

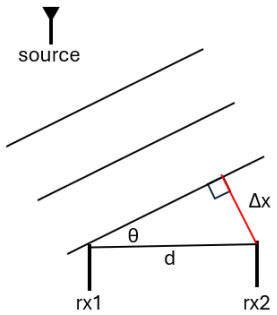


Figure 7: Schematic overview of the antenna array for the phase interferometry direction finding method.

For accurate measurements the distance (d) between the two antennas must be smaller than half the wavelength to avoid spatial aliasing [13]. Due to only having two antennas 180 degree phase ambiguity can occur. This means that one phase result can be caused by a two different signal directions. With spatial aliasing, there would be more phase ambiguity.

4.3.4 Direction finding method analysis

For this project, Watson-Watt is deemed too complex. Pseudo-Doppler needs four antennas and is therefore not practical. Furthermore, this is also too complex for this project. The phase interferometry method is accurate enough since, the direction finding should only indicate in which direction the operator should go, the exact angle is not needed. The phase interferometry device which uses two antennas is the most simple method from the ones discussed and is therefore chosen. Since the system will be used to follow the transmitter, the idea is that the transmitter stays in the antenna bore sight and that the phase ambiguity is not a big problem. Two variations of this technique are used in this thesis.

4.4 Localizing the transmitter

The transmitted signal arriving at the receiver antennas can be schematically represented as seen in Figure 7. Because the signal is arriving at an angle, in order to get to the second receiver antenna the signal needs to travel an additional distance Δx. With this extra distance travelled, the angle of arrival θ can be determined because $sin(\theta) = \frac{\Delta x}{d}$. This can be written as $sin(\theta) = \frac{\Delta t \cdot c}{d}$ using the fact that the wave travels at the speed of light, c, and that $\Delta x = \Delta t \cdot c$. Therefore the second antenna receives the signal with a time delay of Δt in Equation 4 with respect to the first antenna.

$$\Delta t = \frac{sin(\theta) \cdot d}{c} \tag{4}$$

This means that the signals received by the antennas will be as described in Equations 5 and 6.

$$s_1(t) = x(t)e^{2j\pi f_c t} \tag{5}$$

$$s_2(t) = x(t - \Delta t)e^{2j\pi f_c (t-\Delta t)} \tag{6}$$

With f_c the carrier frequency and $x(t)$ the encoded signal. The SDR will down-convert the received signal to base-band this can be seen mathematically as multiplying the signal with $e^{-j2\pi f_c t}$ to do so. This means that after going through the ADC converter s_2 will look like Equation 7.

$$s_2(t) = x[n]e^{-2j\pi f_c \Delta t} \tag{7}$$

With $-2\pi f_c \Delta t$ being the phase difference between the two signals. The two antennas are placed half a wavelength apart so $d = \frac{\lambda}{2} = \frac{c}{f_c \cdot 2}$. Substituting this in Equation 4 gives the phase difference $\Delta\phi = -\pi sin(\theta)$, where $\Delta\phi$ is the phase difference between the two incoming signals. By rewriting this, the angle of arrival can be obtained .

$$\theta = sin^{-1}(\frac{-\Delta\phi}{\pi}) \tag{8}$$

In this derivation the left antenna was taken as the reference. This means that the phase of the left antenna is subtracted from the phase of the right antenna. When a signal comes from the right, the sign of the angle will be changed.

Now the signal still needs to be recognised amongst the noise. This can be done by cross-correlating the received signal with a known reference signal, giving the same result as applying a matched filter. This is described in the next section. The result is a peak at the end of the coded signal, see Figure 8.

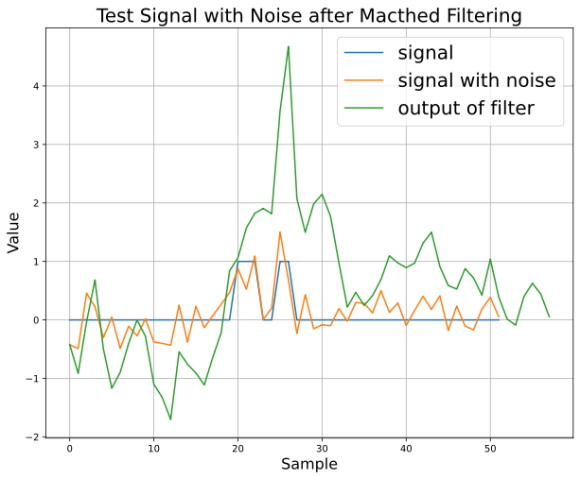


Figure 8: The output of a matched filter on a signal with noise.

Doing this on both of the incoming signal will give the location of the code. The phase of the incoming signal at the location of the peak can then be determined by one of the methods described in Section 6. Taking the difference of the two results and using Equation 8 yields the angle of arrival.

4.4.1 Matched filter

A way to maximize the SNR is by using a matched filter [14]. This filter is described by the transfer function in Equation 9.

$$H(f) = K\frac{S^*(f)}{\mathscr{P}_n(f)}e^{-j\omega t_0} \tag{9}$$

With $S^*(f)$ is the Fourier transform of the known signal, $\mathscr{P}_n(f)$ is the PSD of the input noise and K a nonzero constant. With the assumption of white noise, with $\mathscr{P}_n(f) = N_0/2$, $H(f)$ becomes

$$H(f) = \frac{2K}{N_0}S^*(f)e^{-j\omega t_0} \tag{10}$$

$$h(t) = s(t_0 - t) \tag{11}$$

This is an inverted copy of the known signal and time-shifted with t_0. In this case, with white noise, the matched filter can be carried out by cross-correlating the input with the known signal waveshape [14]. This can be seen with the following derivation.

$$r_0(t_0) = \int_{t_0-T}^{t_0} r(t)s(t)dt \tag{12}$$

Here $r_0(t_0)$ is the output of the cross-correlation of the known signal $s(t)$ and the input $r(t)$. Applying the matched filter to the input gives

$$r_0(t_0) = r(t_0) * h(t_0) = \int_{t_0-T}^{t_0} r(\lambda)h(t_0 - \lambda)d\lambda \tag{13}$$

$$h(t) = s(t_0 - t)\ for\ 0 \leq t \leq T \tag{14}$$

Substituting Equation 14 in 13 give the same result as Equation 12.
Since a convolution in time domain is equal to a multiplication in the frequency domain, the Fourier transform of the result of the filter applies to a signal $r(t)$ with reference signal s(t) is

$$R_o(f) = \mathfrak{F}\{r(t) * s(-t)\} = R(f) \cdot S(f)^* \tag{15}$$

Using this filter, with white noise, results in an SNR described in the equation below.

$$\left(\frac{S}{N}\right)_{out} = \frac{2}{N_0}\int_{-\infty}^{\infty} s^2(t)dt = \frac{2E_s}{N_0} \tag{16}$$

With E_s the energy in the signal.

4.4.2 Gold codes

The frequency bands used for this application are licence free and and can thus be used for multiple applications at the same time. This means that in order to find the right signal, a recognisable code needs to be used. To be recognisable a code needs to have an auto-correlation function that is close to a delta spike at 0-lag and a very low cross-correlation with any other signal and noise.
One optimal version of this is the Gold code. Gold codes or Gold sequences are a type of binary sequences that are used in telecommunication. It has three key properties, it should be periodic, have a low cross-correlation with other signals and random noise and the amount of zero's and ones should be balances. The low cross-correlation means that different Gold codes should not overlap much [15]. Furthermore, they should be pseudo-random. Meaning that they look random, but are still deterministic and reproducible. Figure 9 shows an example of such a gold code.

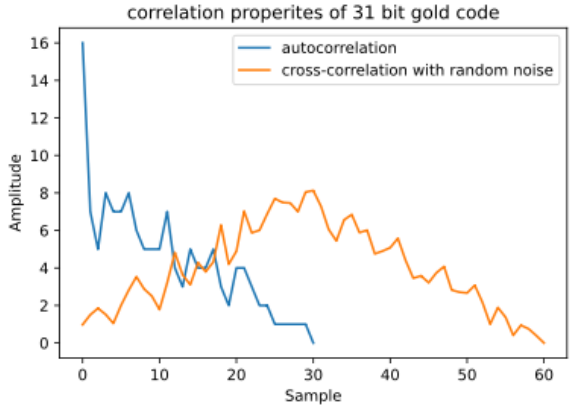


Figure 9: Auto- and cross-correlation of the gold code [1 1 1 0 0 1 1 1 0 1 0 1 1 0 0 1 0 0 1 0 0 1 1 1 0 0 1 0 0 1 0]

4.5 Challenges for direction finding

There are multiple challenges for direction finding. In this section the ones that are the most prominent in this use case are discussed.

4.5.1 Phase ambiguity

When only two antennas are used, 180 degree phase ambiguity can occur. This could be solved by adding a third or even a fourth antenna [13]. Another way of dealing with this problem is by comparing the amplitude of the incoming signal at both the antennas [16]. The direction of the incoming signal could also be induced from previous measurements. Furthermore, at the start of the direction finding, the transmitter is within sight, so the starting position is deduced from that.

4.5.2 Transmitter power output and path loss

Due to the low weight of the hornet, the battery must be small. This causes the signal to noise ratio to be low. On top of that there is also the issue of free space loss and objects obstructing the field of view. The free space loss can be approximated by Equation 17

$$FSPL = (\frac{4 \cdot \pi \cdot d \cdot f}{c})^2 \tag{17}$$

Where d is the distance between the transmitter and receiver. In the case of 433 MHz, this equation gives $FSPL_{500m} = 328.97 \cdot 500^2 = 80dB$.
In any area that is not an open field, the signal will also be heavily attenuated by obstacles, either man-made or natural. The main reasons for power loss are multipath interference, shadowing and absorption, which all have a larger effect with an increasing frequency [17]. It is however difficult to make a standard model of this attenuation as every location is different and rural environments even change with the seasons. One popular model used for path loss in urban environments is the Hata model which is based on the Okumura empirical data [18]. Despite the fact that this application does not exactly fit the requirements of the model, it is still a better approximation than simply taking the free space loss.
The Hata model for path loss is as follows:

$$L_{path} = 69.55 + 26.16 \cdot log_{10}(f) - 13.82 \cdot log_{10}(h_B) - \alpha(h_m) + (44.9 - 6.55 \cdot log_{10}(h_B))log_{10}(d) \quad (18)$$

$$\alpha(h_m) = 0.8 + (1.1 \cdot log_{10}(f) - 0.7)h_m - 1.56 \cdot log_{10}(f) \quad (19)$$

h_b and h_m are the height of the transmitter ans receiver in meters respectfully, f is the carrier frequency in MHz and d is the distance in km. $\alpha(h_m)$ is the correction factor for a medium-small city. Filling in Equation 18 using $h_b = 2m$ and $h_m = 1.5m$ gives a path loss at 500m of $L_{path,hata} = 121.45$ dB.

4.5.3 Multipath

A non-directive antenna will be used for the transmitter, meaning that the signal will be radiated in all directions. Due to reflections from the environment (e.g. houses, trees) the signal can also be received from multiple directions simultaneously, see Figure 10. This multipath problem is a challenge for the direction finding [19]. It creates a copy of the signal that arrives from another angle than the initial signal and with a time delay due to the different path. When calculating the angle of arrival of the incoming signal, these multipath reflections pose a problem. However, due to the fact that they are reflected, their power is reduced and they are delayed. Due to these characteristics, the multipath signals could be filtered out. Directional antennas are less sensitive to multipath interference. However, when a directional antenna is not placed in the direction of the transmitted signal, the received signal will be even lower than with a non-directional antenna. For the purpose of direction-of-arrival systems this means that the range of detection is limited to the beamwidth of the antenna [20].

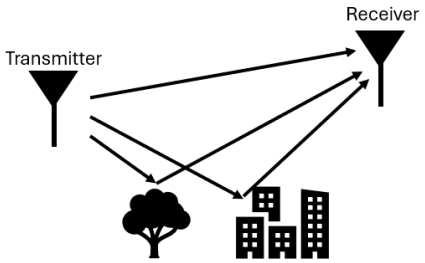


Figure 10: Visual explanation of multipath due to obstacles.

4.5.4 Height difference

The height at which an insect flies depends on the weight of the insect and it also varies with temperature and air pressure [21]. This, and the placement of the transmitter antenna, leads to a change in the polarization of the transmitter. The polarization loss factor can therefore change as this is defined as the following Formula 20.

$$PLF = |\hat{\boldsymbol{p}}_{Rx}^{*} \cdot \hat{\boldsymbol{p}}_{Tx}| \tag{20}$$

Where $\hat{\boldsymbol{p}}$ is the polarization vector. The power of the incoming signal deceases when the polarization is mismatched.

4.5.5 Noise and interference filtering

Kalman filters are recursive constantly updating filters which make real time predictions based on past observations and dynamic models of the system, enabling accurate estimation of the system's state even in the presence of noisy measurements [22]. The Kalman filter uses five equations. The first three make a correction based on the measurement and prediction. The last two make the prediction.

The first equation is the state equation this one is given in Formula 21. This formula estimates the current state by adding the predicted value of the current state to the predicted value of the current state subtracted from the measurement that is multiplied by some constant. This constant is the Kalman gain. The measurement minus the predicted value of the current state is the innovation, also called the measurement residual [23].

$$\hat{x}_{n,n} = \hat{x}_{n,n-1} + K_n \cdot (z_n - \hat{x}_{n,n}) \tag{21}$$

The second equation is the Kalman gain equation. This equation in given in Formula 22 from [23]. In this formula the estimate variance is divided by the estimate variance added to the measurement variance. The result is a value between zero and one.

$$K_n = \frac{p_{n,n-1}}{p_{n,n-1} + r_n} \tag{22}$$

The third equation is the covariance update equation, given in Formula 23 from [23]. This equation uses the Kalman gain to update the covariance. The estimate uncertainty is decreased each filter iteration.

$$p_{n,n} = (1 - K_n) \cdot p_{n,n-1} \tag{23}$$

The fourth equation is the dynamic system model. This model describes the behaviour of the system. The exact formula is given in Formula 24. x_n is the state that is estimated. A is a square matrix, it relates the step at state n-1 to the state at step n, when the process noise is neglected. It is a description of how the state changes between measurements. B is a vector matrix that relates the control input to the state. u_n is the control input. W_n is the model uncertainty or process noise [24].

$$x_{n,n} = A \cdot x_{n,n-1} + B \cdot u_n + w_{n,n-1} \tag{24}$$

The fifth and last equation is the error covariance. This question is given in Formula 25 from [24]. The A is the same matrix as in the system behaviour equation. Q is the process noise covariance. It represents the uncertainty in the process or model.

$$p_n = A \cdot p_{n-1} \cdot A^T + Q \tag{25}$$

4.5.6 Distance estimation

The distance of the transmitter can be approximated using the receiver signal strengths indication. In a study conducted in Pakistan [25], various distance-RSSI measurements are done in an indoor environment using Bluetooth low energy. This study shows an average decrease of 10 dBm for every 1.32 m distance. For distance measurements of the horizontal distance, a third antenna must be used. This third antenna will measure the elevation angle of the transmitter. With this and the

diagonal distance, obtained from the RSSI measurement, the horizontal distance is calculated using the Formula 26.

$$d_{horizontal} = d_{diagonal} \cdot \cos(\varphi) \tag{26}$$

4.6 Precision estimation

4.6.1 Cramer Rao Bound

[26] The accuracy of the measured delay of the antennas could be estimated using the Cramer Rao Bound (CRB). This band indicates the lower bound of the achievable variance of the unbiased estimator [27]. In this case the estimator of the phase difference is unbiased, since there are no differences between the average measurements if they are conducted multiple times in the same environment. The CRB is given by the following Formula 27 [27].

$$CRB(\tau) = \frac{1}{SNR \cdot \beta^2} \tag{27}$$

The accuracy can be found by taking the square root of the CRB. The Signal to Noise Ratio is defined as the total signal power divided by the total noise power as written in Equation 43.

$$SNR = \frac{\frac{S}{N_0}^2}{1 + 2\frac{S}{N_0}} \tag{28}$$

Beta squared is the mean squared bandwidth of the signal and is given by Equation 29 [27].

$$\beta^2 = \frac{\int_0^{T_s} (\frac{ds(t)}{dt})^2 dt}{\int_0^{T_s} s^2(t) dt} = \frac{\int_{-\infty}^{\infty} (2\pi f)^2 |S(f)|^2 df}{\int_{-\infty}^{\infty} |S(f)|^2 df} \tag{29}$$

A second approach for finding the CRB is using the Fisher information matrix J. The CRB is the inverse of this matrix as can be seen in Equation 30 [27].

$$CRB = J_{CR}^{-1} \tag{30}$$

When both the signal and noise power spectral densities are flat, the Fisher information matrix can be written as in Equation 31. In this equation, B stands for the bandwidth of the signal in radians per seconds, T is the time over which the measurements are carried out and ω_c is the carrier frequency of the signal also given in radians per seconds.

$$J_{CR}(\tau) = 2\omega_c^2 \cdot \frac{TB}{2\pi}[1 + \frac{1}{12}\left(\frac{B}{\omega_c}\right)^2] \cdot SNR, \quad for \quad T >> \frac{2\pi}{B} \tag{31}$$

This equation is only valid if the Fourier Series coefficients are independent. This is the case when the measurement time T is significantly greater than 2π divided by the bandwidth. This equation shows that if the observation time increases, the CRB will move downwards leading to a bigger accuracy. Furthermore, a higher carrier frequency results in a higher accuracy [27].

The measurement time is given by the the amount of samples that is received at a time multiplied with the inverse of the sampling frequency as given in Formula 32

$$T = samples_{received} \cdot \frac{1}{F_s} \tag{32}$$

4.6.2 Signal to Noise Ratio

As discussed above, the SNR is important for determining the Cramer Rao Bound. The received signal power is calculated using the effective isotropic radiated power (EIRP). The formula for this is given in Equation 33 from [28].

$$P_{EIRP} = G_{AT} \cdot P_{TX} \tag{33}$$

In this equation, G_{AT} is the transmitted antenna sensitivity and the P_{TX} the power of the transmitted signal. The transmitted antenna gain is calculated using Equation 34 from [28].

$$G_{AT} = \frac{Power\ density\ in\ the\ direction\ of\ the\ maximum\ G(\theta, \phi)}{power\ density\ of\ the\ isotropic\ antenna} \tag{34}$$

For short dipole antennas, this number is approximated by 1.5 [28].
The power density of the transmitted signal is given in Formula 35

$$S = \frac{P_{EIRP}}{4\pi d^2} \tag{35}$$

This transmitted signal is attenuated by the free space loss. This loss is 80dB for a carrier frequency of 433 MHz.
The d in this formula is the distance between the transmitter and the receiver. The received signal power will then be given in Formula 36 from [28].

$$P_{RX} = S \cdot A_{e\_RX} \tag{36}$$

A_{e_RX} is the effective surface area of the receiving antenna. This area is given by Formula 37 from [28].

$$A_e = \frac{G_{RX} \cdot \lambda^2}{4\pi} \tag{37}$$

Therefore the total power formula is as follows, see Formula 38

$$P_{RX} = G_{AT} \cdot P_{TX} \cdot G_{RX} \cdot \frac{\lambda}{4\pi d}^2 \tag{38}$$

The noise power of the system given in Formula 39 from [28].

$$P_{noise} = k \cdot T_{eff\_ant} \cdot B_N \tag{39}$$

In this equation k is the Boltzmann constant, T_{eff_ant} is the equivalent noise temperature and B_N is the bandwidth of the signal. The equivalent noise temperature exists of the sum of the noise temperature all the elements that create noise. For the noise contribution of a device, often the noise figure F is used. In Formula 40 from [28] the relation between the noise temperature and the noise figure is shown.

$$T_e = (F - 1) \cdot T_0 \tag{40}$$

In this equation T_0 is the reference temperature. This is 290 Kelvin.

4.7 Automatic direction finding versus manual direction finding

The current insect tracking devices use a directional antenna, such as a Yagi antenna, to manually find the direction of arrival by turning the antenna and stop at the angle where the signal is greatest. These directional antennas have the advantage of concentrating gain in one direction, which makes the manual direction finding possible. Unfortunately these Yagi antennas are not a good fit for automatic direction finding and low gain omnidirectional antennas will have to be used.
The gain of a Yagi antenna greatly depends on the number of elements and thus on its length. In the case of tracking insects, the antenna must be portable and cannot be to large, therefore for the estimation of the gain, an approximation of 1x1 m will be set for the dimensions. A Yagi antenna consists of 3 different elements, one reflector with a size of $1.05 \cdot \frac{\lambda}{2}$, a $\frac{\lambda}{2}$ dipole and a number of directors each $0.95 \cdot \frac{\lambda}{2}$ of length [29]. For this calculation these elements are spaced 0.25λ apart. The results are laid out in Table 1.

	Element size
Reflector	0.35
Dipole	0.36
Director	0.33
spacing	0.17
Amount of elements	6

Table 1: Measurements (in m) for a portable Yagi antenna at 433 MHz

Using 6 elements goes over the dimension limit, 1.02 m for 433 MHz, but should still be manageable. These dimensions give an approximate gain of 10.5 dB with respect to a dipole, using the table from [29].
In order to increase the range of the tracking system, a directional Yagi antenna can be used for longer distances. However the accuracy of this system heavily depends on the operator, hence for a more accurate short range direction finding, the system described in this report could be utilized.

4.8 User interface

This software is focused on being used by elderly people. This is because the product is initially marketed towards beekeepers. According to an article from 2008 [30], the average age of beekeepers in the Netherlands was nearly sixty years old. Other countries have similar age distributions, in Australia [31] the average age is 52 years old. For the design of the user interface this has to be kept in mind. To create a user accessible and user friendly design, the heuristics of Nielsen are often used in practice. An overview of these heuristics is given here.

4.8.1 10 heuristics of Nielsen

The 10 heuristics of Nielsen show the aspects a design should comply with in order to be user friendly [32].

1. Visibility of system status: The user must know what the system is doing. If the user must wait for something to load for example, a loading bar must be shown.
2. Match between the system and the real world: The system must speak in the same language as the user does. This means that recognisable icons must be used for example.
3. User control and freedom: The user must be able to undo an unwanted action.
4. Consistency and standards: When performing one action, this action must look the same also on other platforms.
5. Error prevention: The system must help the user prevent to make mistakes.
6. Recognition rather than recall: Do not let the user memorise information.
7. Flexibility and efficiency of use: Short cuts must help advanced users with making the task more efficient.
8. Aesthetic and minimalist design: Only relevant information must be shown on the screen.
9. Help users recognize, diagnose, and recover from errors: Error messages must be easy to interpret.
10. Help and documentation: Help users with an explanation option or a tutorial.

5 Program of requirements

5.1 External requirements

The signal that is worked with is produced by a resonator. It has a carrier frequency of 433.96 MHz and a duty cycle of 25% of an hour and a maximum power ERP of 450 mW.

5.2 Stakeholders

There are two types of stakeholders in this project. The internal stakeholders are this project group (consisting of the authors and the other members mentioned in the preface), their supervisors and the university. The external stakeholders are the customers and the government. The government is a stakeholder, since this project must abide the law written up by the government.

5.3 User stories

The system must be capable of the following user tasks. This list only summarizes the tasks that the system must perform in order to give the user a useful product [33]. They are rated using the MoSCoW method. This method gives the user stories a priority. The following user stories with their priorities are developed.

1. Must have: needed in order to have the main function working.
2. Should have: important features that are not vital, but have significant value.
3. Could have: would be nice to have, but could be left out.
4. Wont have: ideas that can be developed at a later stage.

5.3.1 Must haves

- As a user I want to see at which direction I must go, so that I can follow the hornet easily.
- As a user, I want the displayed direction to be updated in real-time, so that I can make timely adjustments to my path.

5.3.2 Should haves

- As a user, I want the system to be portable and lightweight, so that I can carry it easily while moving.
- As a user I want to see an indication of the distance of the hornet, because I don't want it to get out of range.
- As a user I want to be able to start and stop tracking to save the devices energy.

5.3.3 Could haves

- As a user I want to be able to set a stop watch to measure the time it takes to track the nest for future research.
- As a user, I want to log the direction and distance data, so I can analyze it later for patterns or research purposes.
- As a user, I want to have a battery level indicator on the display, so that I can monitor the device's remaining power.

5.3.4 Won't haves

- As a user, I want an audible alert when the signal strength drops below a certain threshold, so I am aware when the hornet is moving out of range.
- As a user, I want to have a weather-resistant design, so I can use the device in various environmental conditions without worry.
- As a user, I want to have a backlight on the display, so I can use the system in low light conditions.

5.4 Requirements

Based on the user stories and the theory, the following set of requirements is developed. After examination only the must haves, the should haves and a couple of could haves can be realized within the given time frame. The requirements that belong to the key performance indicators are marked with a heart symbol.

5.4.1 Functional requirements

♡ Signals with a 433.96 MHz carrier frequency and a bandwidth of 1.5 MHz should be able to be received.

♡ The direction to go to should be displayed on the LCD screen.

- The software defined radio dongle must mix and filter the signal.
- The two antennas of the system must work on the same clock.
- Cross-correlation must be used with a reference signal to obtain the phase delay.
- The Angle of Arrival must be calculated using the phase delay.
- The system must give accurate (less than 10°differences from stationary when moving 20 km/h) results while being moved (every direction).
- The system must give accurate (less than 5 degree difference) when being stationary.
- Using the received signal strength indicator either a distance estimation or a signal reception indication must be given.
- The tracking data must be logged onto the Raspberry Pi's memory.
- The stopwatch should be displayed on the screen.
- The tracking should be able to be disabled to save the system's energy.
- The stopwatch functionality should be able to be disabled.

5.4.2 Non-functional requirements

♡ The length of the omnidirectional dipole antennas must be 35 cm.

- The system latency must be no more than 2.9 seconds. This is how fast a good flying hornet (6 m/s) can get out of the 10° accuracy when at a distance of 100 m.
- The distance between the antennas must be 35 cm.
- Software defined radio must be used.
- The SDR dongles must either have an Ethernet or a USB-A connection.
- The user interface must be intuitive. The direction must be interpretable in 5 seconds.
- The power supply must be a portable power bank.
- The system must be mobile (under 5kg).
- The software must run on the Raspberry Pi 4b.
- The screen must have touchscreen.
- The program must be written in Python.
- Software defined radio must be used.
- The system must work with the RTL dongles.
- The system must cost less than 500 euros.
- The user interface should be clear and easy to read. The buttons need to be at least 1 cm^2 [34].
- The system must have a battery life of at least 10 hours.

6 Methodology

First a system overview is given. This overview shows the different hardware components used and the software that accompanies it. After that, the algorithm to create the angle of arrival is described. Then the various steps of developing the software is discussed. Lastly, the precision calculations are presented.

6.1 Circuit of the receiver

The system is first made using a laptop and the Software Defined Radio (SDR) dongles of the brand RTL. After successful tests, the Raspberry Pi will be used. The laptop is too big to carry around and therefore the eventual product will make use of an Raspberry Pi computer with its 3.5-inch display. This combination fits on the handlebar of the bike and acts like a bike computer. In Figure 11 the circuit with the Raspberry Pi is shown. For hardware setup used for prototyping in this project are shown in Figure 12. Due to a lack of USB-A inputs on the MacBook, a USB hub is used. This hub does feature two USB-A slots, however the width of the dongles causes the second slot to be blocked. To solve this, a second converter is used which converts the USB-A to a USB-C which is then either put into another port of the Macbook or for convenience in the same USB-hub. The latter setup is shown in Figure 13. A second (Windows) laptop featuring multiple USB-A slots was available during the project, but the RTL SDR drivers did not work on this computer. Lastly the antennas that are used are shown, these antennas are 32 cm in length, which is slightly to short for the transmitted signal. However, the alternative setup would have an antenna length of 50 cm. The antenna is shown in Figure 14. Due to budget restrictions, the choice is made to not buy new antennas and use the ones delivered with the RTL dongles.

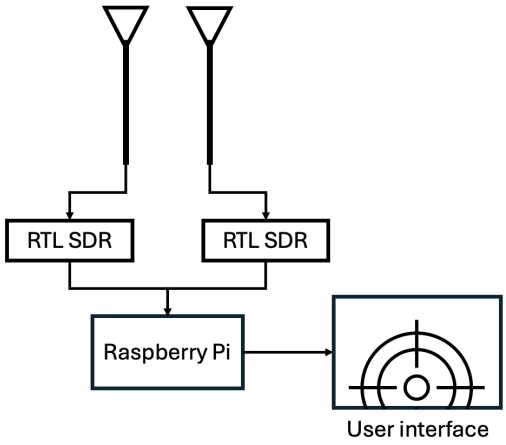


Figure 11: Schematic overview of the receiver circuit.

Figure 12: Figure showing the prototyping hardware setup.

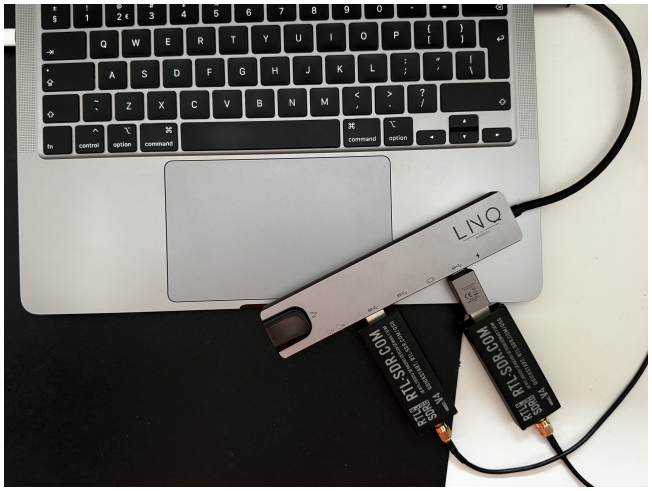


Figure 13: Figure showing the prototyping hardware USB setup.

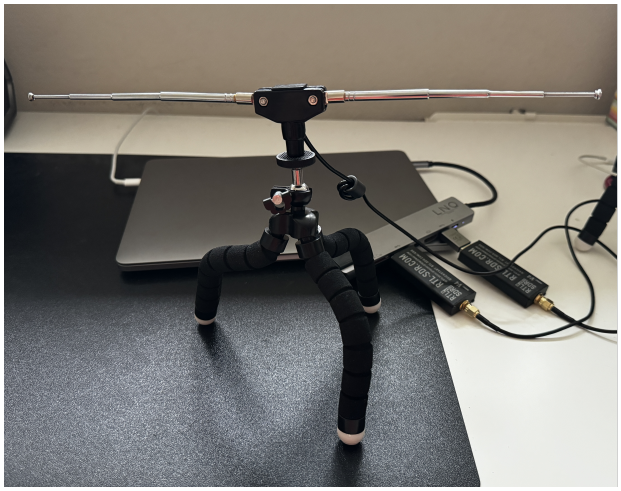


Figure 14: Figure showing the prototyping hardware antenna setup.

6.1.1 Antenna array

The antenna array consists of two omnidirectional antennas. They are placed horizontally next to each other to be able to measure the phase difference (and calculate the time difference) between the

incoming transmitted signal. The distance between these antennas must be either a half or a quarter wavelength. The antennas are mounted on top of a backpack to make the system portable. This way it can be used while walking, cycling and other activities that could be useful for tracking the hornets. This way of mounting method also gives the operator free hands to use the screen or hold other objects.

6.1.2 RTL Dongle

There are various SDR dongles on the market. Considering that the receiver does not have to transmit anything, there is no need for a dongle that is half duplex. Considering the low price of the total receiver, the RTL dongle is the best choice. For this project the RTL SDR Blog v4 was used, which can receive signals from 500 kHz to 1.7 GHz, has a maximum stable sampling rate of 2.4 MS/s and has its own python library. A block diagram of the SDR can be found in Appendix A.

6.1.3 PC type

Since the signals need to be read out, processed and displayed. This can not be done by a micro controller like an Arduino. Therefore a tablet, smartphone or a Raspberry Pi should be used. For using the system on a bicycle, the use of a Raspberry Pi is deemed optimal.

6.1.4 Touchscreen

There are various sized of touchscreens on the market. For this project the 3.5-inch GPIO Raspberry Pi screen is chosen. This screen has approximately the same dimensions as the computer and it fits sturdy due to the GPIO pin connection.

6.1.5 Power supply

The Raspberry Pi needs a power supply to work. Since the whole system needs to be handheld, a power-bank is the ideal solution.

6.2 Explanation of the algorithm

In the following section two methods are attempted. The first one calculates the time difference between the two incoming signals. This time difference is obtained by cross-correlation in frequency domain (multiplication with the flipped complex conjugate) with a reference signal. The outcome of these two multiplications are then divided. The location of the peak of the time domain visualization of this division then gives the time difference between the two arriving signals. Multiplying this time difference with $2 \cdot \pi \cdot f_{code}$ gives the phase difference. Which is then used in Equation 8 to obtain the angle of arrival.

The second approach calculates the phase difference in the frequency domain only.

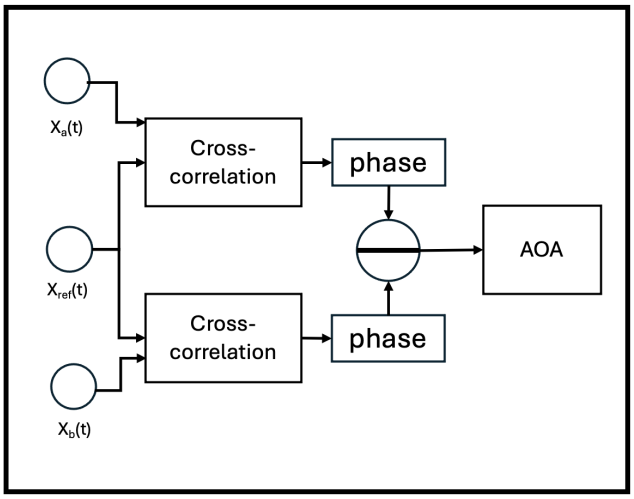


Figure 15: Schematic overview of the direction algorithm

The distance estimation is done using the received signal strength indicator. The signal of the transmitter decreases in strength as the distance from the source increases. Using this, the distance from the transmitter can be estimated.

6.2.1 Programming language

The received data coming from the dongles is processed using the Python programming language. Python is deemed to be the best choice given that it has a lot of signal processing libraries which can be used. Numpy is a powerful and commonly used library for data processing just like Scipy. The RTL SDR also has its own Python library librtlsdr and wrapper pyrtlsdr [35]. For testing purposes, the Matplotlib pyplot class is effective.

6.3 Development of the system

To develop a software efficiently it is required that tests are done for each step in the process. For a structured software application the code is split up into separate classes. Each class has its own responsibility. The entire class structure of the software can be seen in Figure 34. An enlarged version of this figure is given in Section A. The code is given in Section B. For each of the steps it is indicated to which class it belongs.

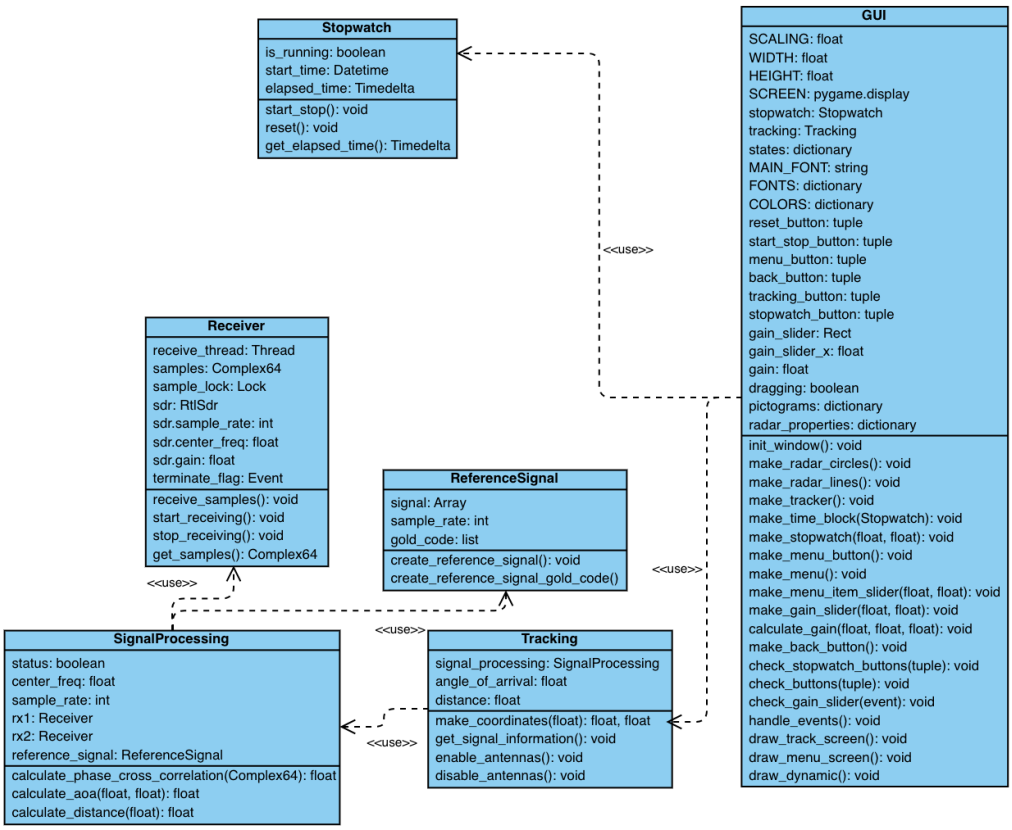


Figure 16: Visual representation of the class structure of the software using Unified Modeling Language

The steps to building the automated direction finding system described in the program of requirements are stated to be the following. The last two steps have not been executed reasons for this can be found in Section 8.

1. Create Graphical User Interface.
2. Receive the signal with one antenna.
3. Make a phase difference calculation function with a reference signal.
4. Receive the signal with two antennas.
5. Make a function for calculating the phase difference of two signals.
6. Calculate the phase difference of the received signal using the function.
7. Calculate the phase difference between the two antennas.
8. Calculate the angle of arrival with the phase difference and show this on the GUI.
9. Calculate the distance and show this one the GUI.
10. Improve the software using cold codes.
11. Integration with the total system.
12. Improve the software using a Kalman filter.

Two algorithms for calculating the phase difference have been proposed. The first one uses the time difference of arrival and the second uses the difference of the phase of the FFT's.

6.3.1 The graphical user interface

Before the dongles were available, the graphical user interface was made. This GUI has its own class called GUI B. This GUI is developed using the Pygame library and the Numpy and Datetime

libraries. It also imports the Tracking and Stopwatch class. The Stopwatch class B is responsible for the stopwatch functionality. The layout visualizes a radar interface. This interface is commonly known, so the idea is that it is easily interpretable. The color palette can be seen in the following Figure 17. These colors are chosen because of the similarities to radars. In red and green usage in the menu is an intuitive way of showing on and off.

Figure 17: The colors used for the user interface. The color codes left to right: 08090A, FFFAF, 32CD32, F0544F

The layout is first made in the Figma [36]. This is an online interface design tool. The result is seen in Figure 18. In the first sketch, the options are turned on. The second sketch, seen in Figure 33 in the appendix shows the interface when the options are off. A middle ground when either of the option is enabled is also possible.

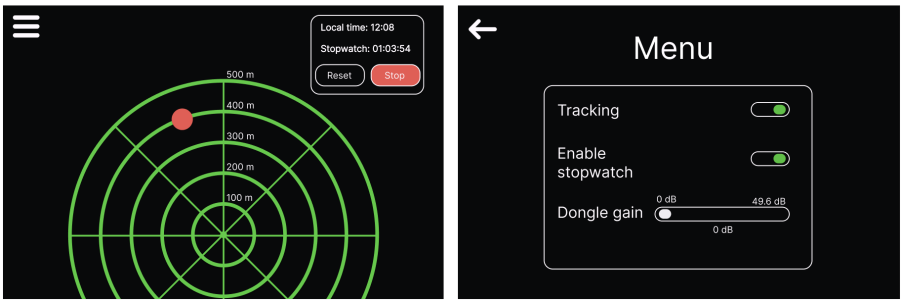


Figure 18: Figma sketch of the user interface when the menu options are turned on.

The system has a minimalist design. There are only a few buttons the user can press. The stripe button leads to the menu. In this menu, the tracking and the stopwatch can be put on or off. When the tracking is disabled, the red dot indicating the tracker position is removed. When the stop watch is disabled, only the time is shown on top of the screen. To make the software accessible for elderly people, a stylus is included. The people from this generation have not been brought up using touchscreen. Users who prefer the touchscreen can still use it.

6.3.2 Receiving a signal

For receiving signals using the RTL SDR dongles a Receiver class B is made. This class creates the RTL objects using the RtlSdr library and receives the samples from the dongles using a separate thread from the Threading library to enable this event to happen simultaneously with the presentation of the user interface. It also uses the Numpy library to save the sample data into Numpy complex64 arrays. To test this class a 433 MHz RF Transmitter, an Arduino and a coil antenna are used. This Arduino is powered by a 9V battery. The test signal is given in Figure 19. The time plot generated from the SDR output signal by the test code C is given in Figure 20. This figure shows that the transmitted signal is well received although noise is clearly present. The coil antenna, free space loss, and other factors contribute to this. The Arduino signal is a repeated signal consisting of two 30 ms high bits separated by a delay of 30 ms. The signal is received in an indoor environment on a distance of 2.5 meter.
It can be seen from Figure 20 that this is not an exact copy of the reference signal. The peaks are not all spaced equally apart. This may be explained by the fact that the transmitter used is a cheap and imperfect device.

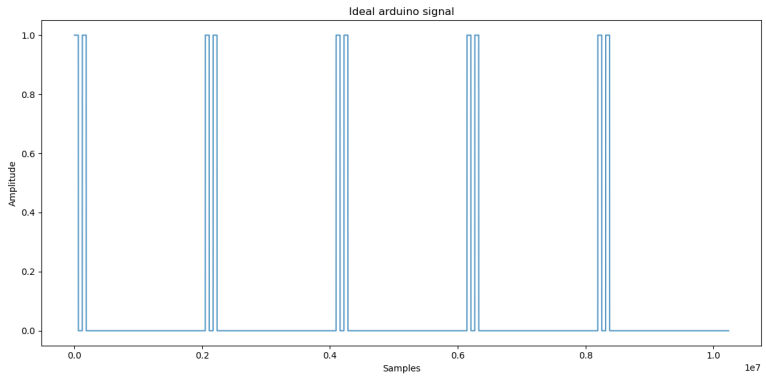


Figure 19: Visualisation of the normalized transmitted Arduino signal given in samples.

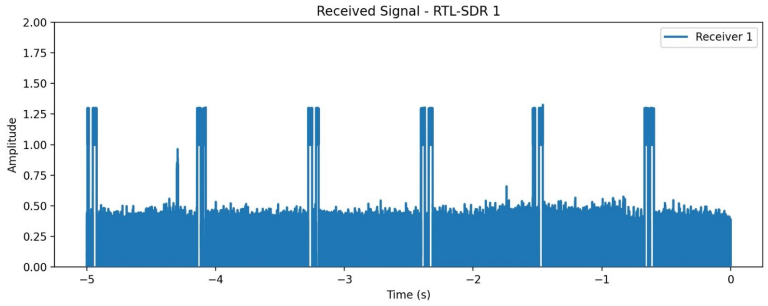


Figure 20: Time plot of the received signal from one RTL SDR dongle.

The figure shows the received data from right to left. This is because it is programmed as an animation. The -5 indicates looking into the past. The reason for making an animation and taking a screen shot of that is because it takes some time for the dongles to be instantiated and produce usable data. The exact timing differs and can therefore not be hard coded. The amplitude shown in the plot is the output of the dongle. This output has no unit for the amplitude because the dongle normalizes it. It is unnecessary to know the exact value of the received signal strength at this point as long as the signal can be well distinguished from the noise, which is the case here.

6.4 Receive two signals

In order to obtain samples from two dongles, a second object of the receiver class is made. For this to work, the serial number of the second dongle is altered. Figure 21 shows the captured signals of the two antennas using the test code C.

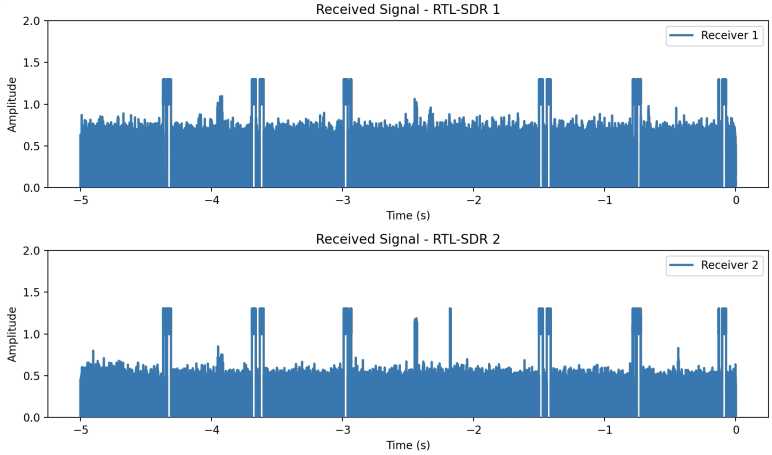


Figure 21: Plot of the two received signals using the two dongles.

This figure shows two different received signals which means that the two antennas and dongles are properly instantiated. The amount of noise captured by the two antennas differs. A possible reason for this is the placement of the antennas. This test is conducted indoors which means that other (household) equipment might cause interference. Due to the low range of these sources, one antenna might receive more of these signals than the other.

6.4.1 Phase difference function

The next step is to create and test a function that calculates the phase difference of two signals using cross-correlation. One signal is the received signal from the antenna and the other is a repeated reference signal. This function is part of the SignalProcessing class B. The reference signal is made in the ReferenceSignal class B such that this signal can easily be changed. This way of programming adheres the open-closed principle. The phase calculation function is tested using a hardcoded version of the Arduino output. The shifted signal and the reference signal are shown in Figure 22. This figure shows the time shifted signals and the calculated phase difference. The Arduino signal is 1 second long, so a delay of 0.05 seconds correctly corresponds to a phase delay of 18 degrees for the coded signal.

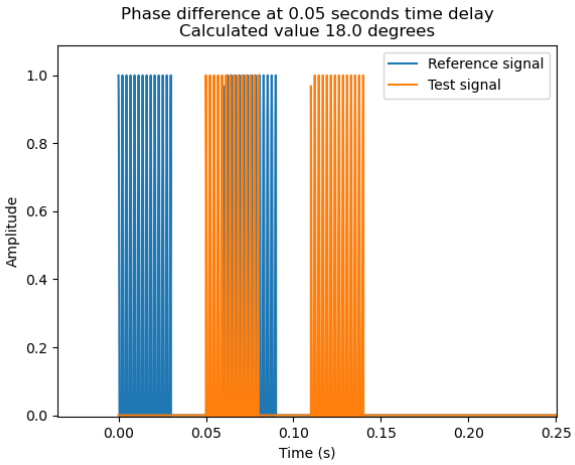


Figure 22: Plot showing two artificially made signals and their time and calculated phase difference.

The second step is to add random noise to the shifted signal and see whether it gives an accurate result. This result is given in Figure 23. The calculated phase difference is still the expected value. The code of these two test can be found in Section C.
In this algorithm, both the reference signal and the received samples are zero padded to obtain a length of both lengths - 1 and are then cross-correlated. The reference signal is made by using the ideal Arduino code and modulating it with a carrier sinusoidal signal of 434 MHz. The cross-correlation is done by multiplying the fast Fourier transform (FFT) of the zero padded received signal with the flipped version of the FFT of the conjugate of the zero padded reference signal. The inverse fast Fourier transform is then taken of the multiplicand. Then the index of the peak is found and this location corresponds with the phase difference.

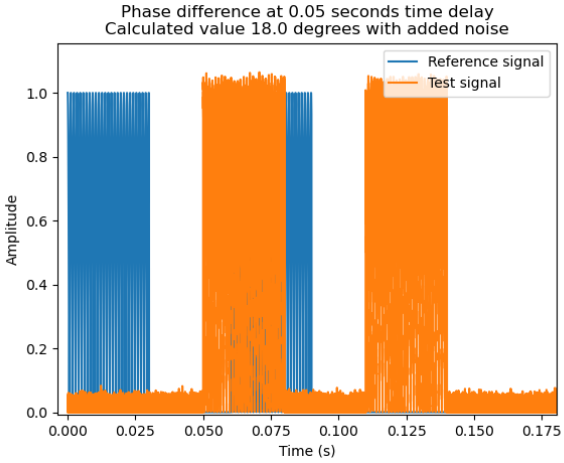


Figure 23: Plot showing two artificially made signals and their time and calculated phase difference with added random noise.

6.5 Phase difference calculation with the received signal

After the working of both the receiving software and the phase-difference calculation software is validated, they are combined. The transmitted signal discussed in the receive signal test is again used. It is expected that the calculated phase difference of the reference signal and the received signal should look like a sinusoid. The reference signal does not change, but the received signal does. This signal is a moving wave and if at each time the phase difference is calculated, the movement of the signal should be shown in the phase difference. In Figure 24, the received signal and the calculated phase difference are shown. The code for this test is given in Section C. As can be seen in the figure, it is not an ideal sinusoid. An analysis of this can be read in Section 7.1.

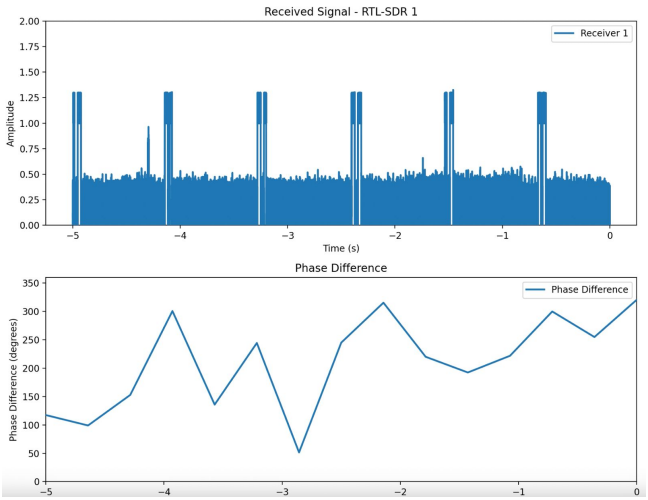


Figure 24: Figure showing the signal received from the dongle (upper plot) and the calculated phase difference with the reference signal.

6.6 Phase difference between two antennas

After the two signals are received successfully, the phase difference of these compared to the reference signal signals is calculated. The phase difference between the two received signals is then calculated using the calculate_delta_phi function in the SignalProcessing class. To test this function, a second shifted function is added to the phase difference test described in Section 6.5. This third signal and the calculated phase difference between the two shifted signals can be seen in Figure 25.

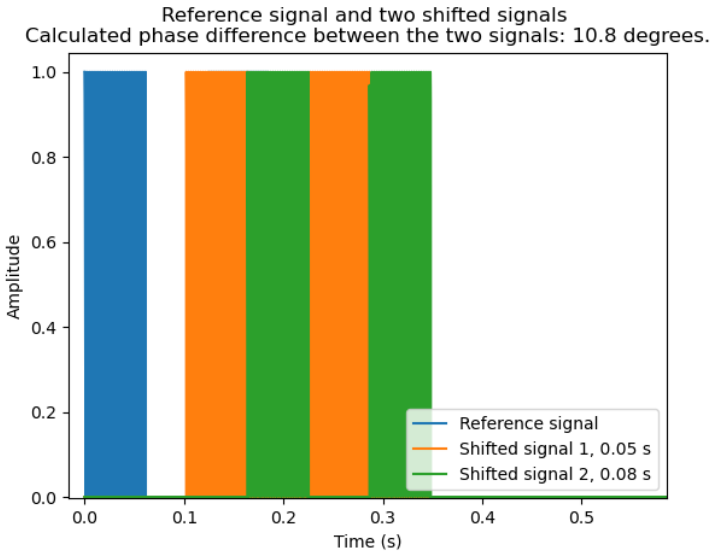


Figure 25: Plot showing three artificially made signals and their time and calculated phase difference.

When noise is added to both shifted signals, the outcome remains the same as can be seen in Figure 26.

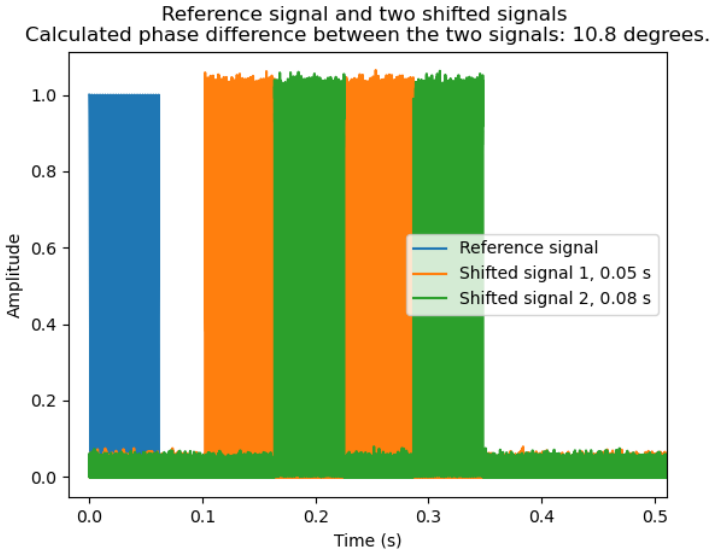


Figure 26: Plot showing three artificially made signals and their time and calculated phase difference with added random noise.

The code for these tests can be found in Section C. This algorithm uses the same start as the algorithm for finding the phase difference between one received signal and the reference signal. This multiplication is executed twice, once for each received signal. These products are then divided. The phase difference between the received signals is found at the location of the peak of the IFFT of the outcome of the division. This phase difference should remain constant when the transmitter is not moved. A time plot of the received signal and the phase difference is shown in Figure 27. Due to computational load, only one of the received signals is shown. This upper plot shows that the signal is well received, which is mandatory to know that the receivers are correctly instantiated. The phase difference in the second plot is not constant. This problem is discussed in Section 7.2. The lack of measurement points in the plot is explained in Section 7.1.

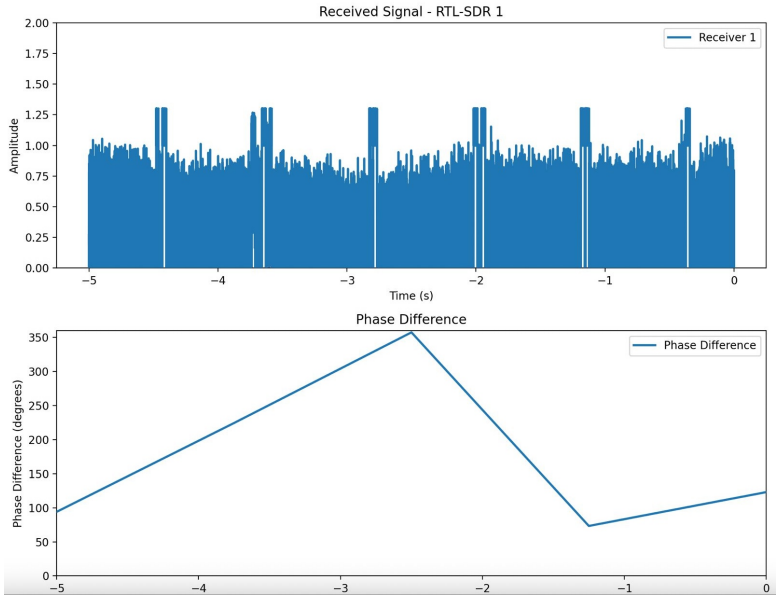


Figure 27: Plot of the received signal and the phase difference between the received signals using the two dongles.

6.7 Angle of arrival calculation

Without reliable phase difference calculations, the angle of arrival cannot be calculated. When using the theoretical value 10.8 degrees obtained in Section 6.6, the angle of arrival is calculated using Equation 8. This gives an angle of arrival of 3.4 degrees. First 10.8 degrees is converted into radians, then it is plugged into the formula and lastly this radian angle is converted in to degrees.

6.8 Received signal strength indication and distance measurement

As could be seen in Section 4.5.6, the distance can be calculated from the received signal strength indication (RSSI). Multiple experiments using different algorithm have been conducted, but no experiment has shown a correlation between the distance between the transmitter and the receiver and the RSSI. Figure 28 shows the failed experiment. The walking pace of about 4km/h might not have been constant (but definitely between 3 and 5 km/h) during the experiment nor the elevation, but even with these factors a slight decline in received signal strength would be expected. The distance covered in this experiment was around 80 meters until the receiver lost the signal. The code for this test can be found in Section C.

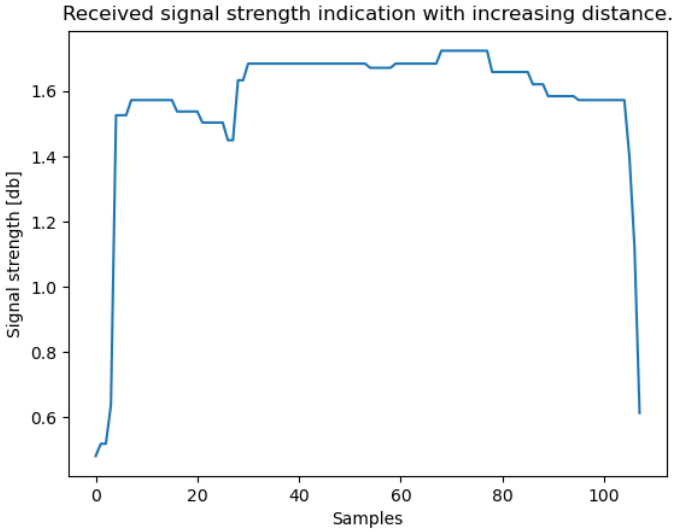


Figure 28: Plot of the calculated received signal strength indications whilst walking away with the transmitter.

One thing that could be measured was the range of the system. The sudden decline in the plot shows the moment where the receiver lost the signal. This was measured (using a smartwatch) to be around 80 meters. This was measured in an urban environment. An analysis of the failed experiment is given in Section 7.4.

6.9 Direct phase difference

A second approach to find the angle of arrival is done using the phase of the Fourier transform of the incoming signals. Here the phases of the FFT at the dominant frequency are subtracted from each other. This method is explained in the following sections. The code for this approach can be found in Appendix C.

6.9.1 Code Identification

The frequency band that will be used is open for use for every one, thus there will be a lot of interference for other signals. Additionally the transmitter is not transmitting continuously, therefore the encoded signal needs to be identified. This is done using a matched filter, a more theoretical explanation is given in Section 4. Figure 29 shows the result of the filter.

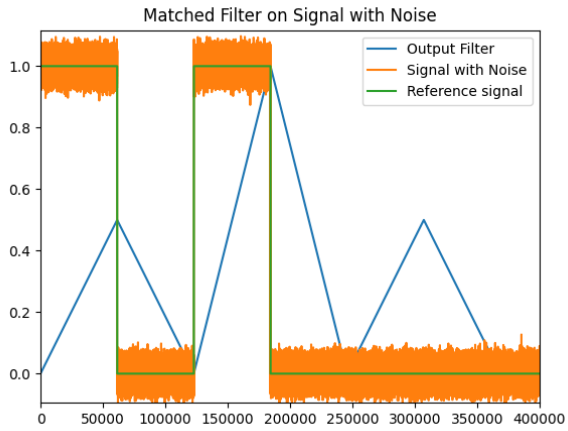
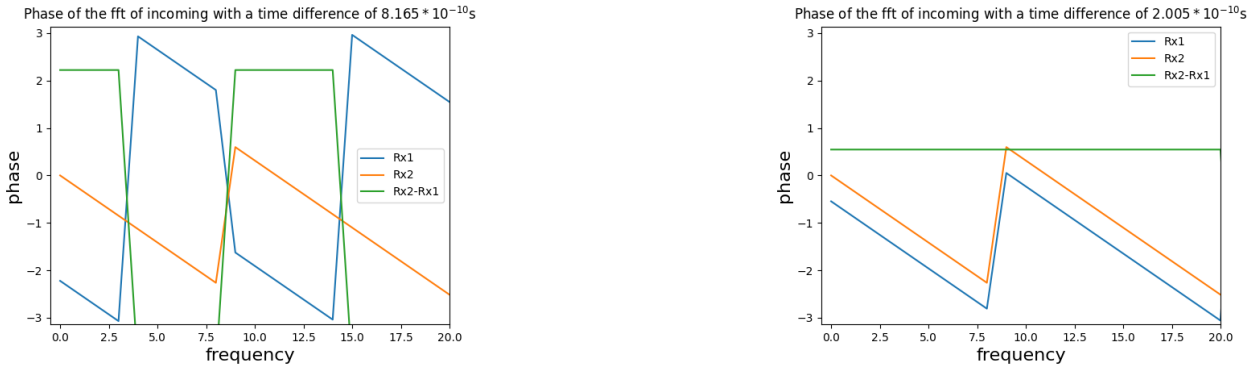


Figure 29: Result of a matched filter applied to a signal with noise

After the code is identified, only the samples of the codes are taken to minimize the amount of noise in the final calculations.

6.9.2 Phase Difference

With the location of the code found, the phase difference can be calculated. This is done in the frequency domain using the fast Fourier transform. Figure 30 shows the result of this operation for two different time delays between the signals.



(a) time delay: $8.165 \cdot 10^{-10}$ **(b)** time delay: $2.005 \cdot 10^{-10}$

Figure 30: The phase of the fft of incoming signals at different time delays

As seen in Figure 30a this method does not give a stable phase difference for all frequencies, however when looking at the dominant frequency of 0 Hz, a correct result can be found. Indeed, Equation 7 shows that a time difference of $8.165 \cdot 10^{-10}$ s and $2.005 \cdot 10^{-10}$ s results in a phase difference of 2.22 rad and 0.5 rad respectfully. As seen in Section 4, to find the angle of arrival the phase difference needs to filled in Equation 8.

6.10 Gold code improvement

The code which is transmitted is of interest for the gain of the receiver and thus the signal to noise ratio. The chosen code adheres to the requisites discussed in Section 4.4.2. The gold code consists of the following bits that are sent using on-off keying: [1 1 1 0 0 1 1 1 0 1 0 1 1 0 0 1 0 0 1 0 0 1 1 1 0 0 1 0 0 1 0]. The transmitted signal has a duty cycle of 25%. It will be on for 0.5 seconds and off for 1.5 seconds. The 31 bits should fit in the on period. The bit duration will therefore be 16.13 milliseconds. The code is visualised in Figure 31. Using this particular gold code in the first phase difference calculation would increase the accuracy.

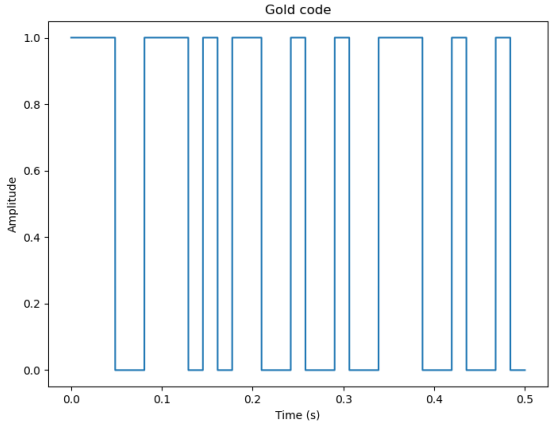


Figure 31: Visual representation of transmitted gold code

7 Analysis of the system

7.1 Algorithm and Python drawbacks

As could be seen in Section 6.5, the calculated phase differences do not correspond with the expectations. After further inspection, the program is to blame for this. During the execution of the program, the calculation time is measured. This elapsed time is taken as the real time difference between when the function is called and the output is returned. As can be seen in Table 2, this difference is substantial and not constant. This renders the output to be unusable.

	Iteration 1	Iteration 2	Iteration 3	Iteration 4	Iteration 5	Iteration 6
Elapsed time (s)	11.7	7.2	6.2	5.8	5.9	21.8

Table 2: Elapsed time of the phase difference algorithm.

For the calculation using two dongles, this calculation time is even worse, due to the added computations. The elapsed time of this algorithm is measured with the same approach as described above. The results can be seen in Table 3.

	Iteration 1	Iteration 2	Iteration 3	Iteration 4	Iteration 5	Iteration 6
Elapsed time (s)	18.6	15.5	15.8	15.2	28.6	15.8

Table 3: Elapsed time of the phase difference algorithm.

The elapsed time in both algorithms is larger than the time axis shown in the figures. It is therefore odd that multiple measurement points are shown in the plot. A possible explanation for this is that the time axis is not correct anymore, since the plot freezes when phase is calculated.

7.2 Synchronization of the dongles

The phase difference calculated in Section 6.6 is not constant. This is caused by the sequential get_samples() call. This causes a time delay between the saved samples of the two dongles. Just like in the previous section, a time analysis is done to measure the problem. In Table 3 the time between the storage of the samples from the first dongle and the storage of the samples of the second dongle is analyzed.

	Iteration 1	Iteration 2	Iteration 3	Iteration 4	Iteration 5	Iteration 6
Elapsed time (s)	$9.5\ k \cdot 10^{-7}$	$2.9 \cdot 10^{-6}$	$1.2 \cdot 10^{-6}$	$3.1 \cdot 10^{-6}$	$5.9 \cdot 10^{-7}$	$1.7 \cdot 10^{-6}$

Table 4: Elapsed time between the storage of the samples of the two receivers.

To synchronize the two receivers, it is possible to remove the crystal from one of the dongles and connect the two using a wire. By having the two dongles work on the same clock frequency, the sample rates and the local oscillator frequencies are made equal. However, the actual sampling and the local oscillators could be out of phase. This means that the calculated angle of arrival is dependent on the this phase difference. In order to counter this problem, various solutions are proposed. In the solution proposed by Tatu Peltola [37], a calibration signal is sent to both dongles by placing a white noise generator directly at the input of both dongles. The pulses of white noise are then cross-correlated to find the phase offset.

7.3 Precision estimation of the algorithm

The estimation of the accuracy of the time difference tau in the two antennas is calculated using second method for the Cramer Rao Bound explained in Section 4.6.1. For this an estimation of the Signal to Noise Ratio is needed. The transmitted signal power is -43 dB as obtained from the thesis of Sofya Mikhaylitskaya and Emmad Hassan [38]. The calculated path loss is 121.45 dB, see Section

4.5.2, therefore the total signal power is -164.45 dB. The noise power is calculated using Equation 39. The noise figure of the RTL dongle is 8 dB [39] this is equal to 6.31 linear. With a bandwidth of 1.5 MHz, the noise power becomes 3.18 $\cdot 10^{-14}$ W, which is equal to -134.97 dB. The SNR used in the Cramer Rao Bound calculation is found using Equation 43 and results in $SNR = 1.267 \cdot 10^{-6}$. The measurement time is calculated using Formula 32. The amount of received samples at a time is 262144, this gives a measurement time of 0.128 seconds. The J_{CR} is then calculated to be 3.603 $\cdot 10^{18}$. Which gives a CRB of 4.620 $\cdot 10^{-19}$. This is the variance of the unbiased estimator, taking the square root gives the accuracy in seconds. Thus, $\Delta t = 5.268 \cdot 10^{-10}$ s. The exact derivation is written in Appendix D. With this minimum time variation, using Equation 4, a minimum angle variation of 0.5° can theoretically be found.

7.4 Received signal strength indication

The failure of the experiment shown in Section 6.8 can be blamed on the lack of understanding of the dongle. The hypothesis is that the dongle automatically corrects for the decreasing signal strength by changing the gain.

7.4.1 Range

The tested range using the Arduino setup is 80 meters. The exact transmitted power is not measured, but given that the output of the transmitter that is attached to the hornet has a lower power, the required 500 meter range will not be achieved.

7.5 Testing the user interface

The user interface is tested by making an analysis based on the Heuristics of Nielsen. The system status is not being shown at the moment. It is expected that the system does not require the user to wait. The pairing of the transmitter needs to be implemented still. The match between the system and the real world is apparent due to the radar interface. The user is able to undo an unwanted action because the menu has the option to switch the various menu options on and off. The action buttons have similar looking icons compared to other platforms. Error prevention is not done yet. The user interface is rather simple and the user does therefore not have to remember anything except for the menu and return icons. Due to the simplicity, there are no possible shortcuts. The user interface does is minimalistic however no error messages are implemented. To help users get familiar with the product, a tutorial of the device will be put online and provided with each purchase.

8 Discussion

For our solution to work according to the requirements, various aspects need to be addressed. Section 7 analyses the encountered difficulties. In summary, a multichannel device and a faster algorithm computer combination is needed. The initial plan was to prototype using the USRP N210 device. With this device, the synchronization of the antennas would not be an issue, since it has multiple channels. Due to a lack of working Ethernet switches, however, we were not able to continue prototyping with this device. In hindsight, working with other SDR's (such as LimeSDR or KrakenSDR) would have been a better option for realizing a working prototype, since these also have multiple channels. This would however surpass the budget of both the client and the university. As is written in Section 7.4, there was not enough understanding of the working of the SDR dongles nor the Python library that accompanies it. This has led to a lot of time and effort spent on trying to understand the open source materials.

Even if the prototype had worked, there is no guarantee that the total system would be a success given the low power output of the transmitter and the current measured range of 80 meters. Furthermore given the nature of the Asian Hornet, it is able to fly across rivers, lakes et cetera whilst flying with a speed of about 21 k/m. The insect would get out of range too fast for the hornet hunter to track it over such a natural obstacle. Also the height at which the hornet flies decreases the horizontal range (using Pythagorus' Theorem), this also makes calculating the angle of arrival more difficult. For exemple when it flies directly about the operator.

In Section 6.1, multiple shortcomings in the hardware are discussed. All of these imperfections lead to the system not working perfectly. The calculation duration of the algorithm is deemed to be the most prominent problem at the current stage of the prototype. Solving these hardware problems would not solve this problem (a faster laptop might, but no research was done on this). After solving the algorithm problem and the synchronization problem, the accuracy should be improved by purchasing new and specialized hardware.

The theoretical precision of the system discussed in Section 7.3, is in practice not feasible. The interference with other devices just as path loss would decrease the precision.

9 Conclusion

After research was done about multiple automatic direction finding methods, an attempt was made to design prototype using phase interferometry. Due to the high computation time and the lack of synchronization, the prototype does not work as intended. The user interface has been fully implemented in Python and if the above stated problems are resolved, almost all the requirements would have been met. Except for the accuracy requirement as they are not examined. Also the distance indication and the operation range of 500 meters is not realized. Furthermore, the system is not tested on a Raspberry Pi 4B since the system did not work on a laptop.

However, the proposed prototype used software defined radio and the cheap RTL dongles to receive the 433.96 MHz signal. It shows the direction to go, the angle of arrival, which is calculated using cross-correlation on the LCD screen and it saves this data. The tracking and the stopwatch can be disabled to save energy.

10 Future work

For future work a more sophisticated SDR dongle will be used. The KrakenSDR for example is known for its usage in automatic direction finding. When such a SDR dongle is used, research about multi path complications can be done. Furthermore, research into using automatic direction finding with a low power transmitted signal can be conducted, as well as the correction of the estimation using a Kalman filter.

11 Bibliography

References

[1] S. Lioy, D. Laurino, R. Maggiora, *et al.*, "Tracking the invasive hornet vespa velutina in complex environments by means of a harmonic radar," *Scientific Reports*, vol. 11, 1 Dec. 2021, ISSN: 20452322. DOI: 10.1038/s41598-021-91541-4.

[2] L. Electronics, *Lowlandtag manual en*$_2$, Belgium tracking device: https://lowland-electronics.be/.

[3] lotek, "Vhf-avian-tags-for-smaller-species-spec-sheet,"

[4] Admin, *rtl-sdr.com*, Apr. 2024. [Online]. Available: https://www.rtl-sdr.com/.

[5] F. W. H. Roel Schiphorst and C. H. Slump, *The front end of software-defined radio: Possibilities and challenges.*

[6] A. about circuits, *Practical Guide to Radio-Frequency Analysis and Design.* [Online]. Available: https://www.allaboutcircuits.com/textbook/radio-frequency-analysis-design/radio-frequency-demodulation/understanding-quadrature-demodulation/.

[7] *Beamforming & DOA — PySDR: A Guide to SDR and DSP using Python.* [Online]. Available: https://pysdr.org/content/doa.html.

[8] A. Karki, *Radio direction finding using pseudo-doppler for uav-based radio direction finding using pseudo-doppler for uav-based animal tracking animal tracking*, 2019. [Online]. Available: https://scholarworks.gvsu.edu/theses.

[9] S. Ready, *The Doppler Effect: Explained with Examples – HSC Physics.* [Online]. Available: https://sciencecready.com.au/pages/dopplers-effect.

[10] "A doppler radio-direction finder," 2001. [Online]. Available: https://api.semanticscholar.org/CorpusID:4677299.

[11] N. Ahmed, "Radio direction finding, theory and practices," 2016. DOI: 10.13140/RG.2.1.1019.4002/1. [Online]. Available: https://www.researchgate.net/publication/289779492.

[12] W. Read, *Review of conventional tactical radio direction finding systems.*

[13] "Interferometer angle-of-arrival determination using precalculated phases," *Radio Science*, vol. 52, pp. 1058–1066, 9 Sep. 2017, ISSN: 1944799X. DOI: 10.1002/2017RS006284.

[14] L. Couch, *Digital and Analog Communication Systems* (Prentice-Hall international editions). Pearson, 2013, ISBN: 9780132915380. [Online]. Available: https://books.google.nl/books?id=j1UjygAACAAJ.

[15] D. S. B. Vaishnavi, "Gold Code Generators: - Dasika Sri Bhuvana Vaishnavi - Medium," Feb. 2024. [Online]. Available: https://medium.com/@dasikavaishnavi/gold-code-generators-d784e2b87984.

[16] J. P. Balsamo, *Phase and amplitude interferometry based radio frequency phase and amplitude interferometry based radio frequency direction finder direction finder.* [Online]. Available: https://scholars.unh.edu/honors/450.

[17] A. G. Longley, *Radio propagation in urban areas.*

[18] M. Hata, "Empirical formula for propagation loss in land mobile radio services," *IEEE Transactions on Vehicular Technology*, vol. 29, no. 3, pp. 317–325, 1980. DOI: 10.1109/t-vt.1980.23859.

[19] *Multipath propagation in v/uhf direction finding systems comparison of 7-channel and 5-channel direction finding systems with correlative interferometer [ci] and vector matching [vm® plath] whitepaper 2.*

[20] R. Sanudin, N. H. Noordin, A. O. El-Rayis, N. Haridas, A. T. Erdogan, and T. Arslan, "Analysis of doa estimation for directional and isotropic antenna arrays," in *2011 Loughborough Antennas & Propagation Conference*, 2011, pp. 1–4. DOI: 10.1109/LAPC.2011.6114043.

[21] J. A. Helms, A. P. Godfrey, T. Ames, and E. S. Bridge, "Predator foraging altitudes reveal the structure of aerial insect communities," *Scientific Reports*, vol. 6, Jun. 2016, ISSN: 20452322. DOI: 10.1038/srep28670.

[22] N. Murugendrappa, A. G. Ananth, and K. M. Mohanesh, "Adaptive noise cancellation using kalman filter for non-stationary signals," vol. 925, IOP Publishing Ltd, Oct. 2020. DOI: 10.1088/1757-899X/925/1/012061.

[23] *Kalman filter from the ground up second edition*. 2023, ISBN: 9789659312016.

[24] O. Cadet, *Introduction to kalman filter and its use in dynamic positioning systems introduction to kalman filter-application to dp dynamic positioning conference*.

[25] F. Subhan, A. Khan, S. Saleem, *et al.*, "Experimental analysis of received signals strength in bluetooth low energy (ble) and its effect on distance and position estimation," *Transactions on Emerging Telecommunications Technologies*, vol. 33, 2 Feb. 2022, ISSN: 21613915. DOI: 10.1002/ett.3793.

[26] I. W. Peshkov, N. A. Fortunova, and I. N. Zaitseva, "Minimizing the cramer-rao lower bound for antenna arrays with directional radiators for direction-of-arrival estimation of radio signals," Institute of Electrical and Electronics Engineers Inc., 2022, ISBN: 9781665470643. DOI: 10.1109/SYNCHROINFO55067.2022.9840952.

[27] B. Sadler and R. Kozick, "A survey of time delay estimation performance bounds," Institute of Electrical and Electronics Engineers (IEEE), Oct. 2006, pp. 282–288. DOI: 10.1109/sam.2006.1706138.

[28] D. J. G. Janssen, *Telecommunications a (ee2t11) lecture 3*, 2023.

[29] E. Notes, *Yagi antenna theory: Yagi antenna basics*. [Online]. Available: https://www.electronics-notes.com/articles/antennas-propagation/yagi-uda-antenna-aerial/theory.php.

[30] J. D. Rob Nijman, *De honingbij hoort erbij*, 2008.

[31] *Beekeepers — Labour Market Insights*. [Online]. Available: https://labourmarketinsights.gov.au/occupation-profile/beekeepers?occupationCode=121311.

[32] J. Nielsen, *Enhancing the explanatory power of usability heuristics*.

[33] A. Business, *Chapter 15: Requirements and user stories*. [Online]. Available: https://www.agilebusiness.org/dsdm-project-framework/requirements-and-user-stories.html.

[34] K. Dandekar, B. I. Raju, and M. A. Srinivasan, "3-d finite-element models of human and monkey fingertips to investigate the mechanics of tactile sense," *Journal of Biomechanical Engineering*, vol. 125, pp. 682–691, 5 Oct. 2003, ISSN: 01480731. DOI: 10.1115/1.1613673.

[35] *pyrtlsdr*. [Online]. Available: https://pyrtlsdr.readthedocs.io/en/latest/Overview.html.

[36] *FigMa: The Collaborative Interface Design Tool*. [Online]. Available: https://figma.com/.

[37] Tejeez, *GitHub - tejeez/rtl*$_c$*oherent : SynchronizedRTL − SDRreceiversanddirectionfinding*. [Online]. Available: https://github.com/tejeez/rtl_coherent.

[38] H. E. Mikhaylitskaya S, *Transmission and reception of an ultra-low power signal*.

[39] D. .-. S. Linux Philip Collier Ab9il, *RTL-SDR Sensitivity Improvement — Linux for Shortwave Listeners — RTL-SDR — KiwiSDR*. [Online]. Available: https://skywavelinux.com/rtlsdr-sens.html#:~:text=The%20RTL%2DSDR%20has%20a,in%20terms%20of%20manufacturing%20tolerances..

[40] Realtek, *Rtl2832u dvb-t cfdm demodulator+usb 2.0 datasheet*, 2010.

A Additional figures

A.1 RTL-SDR

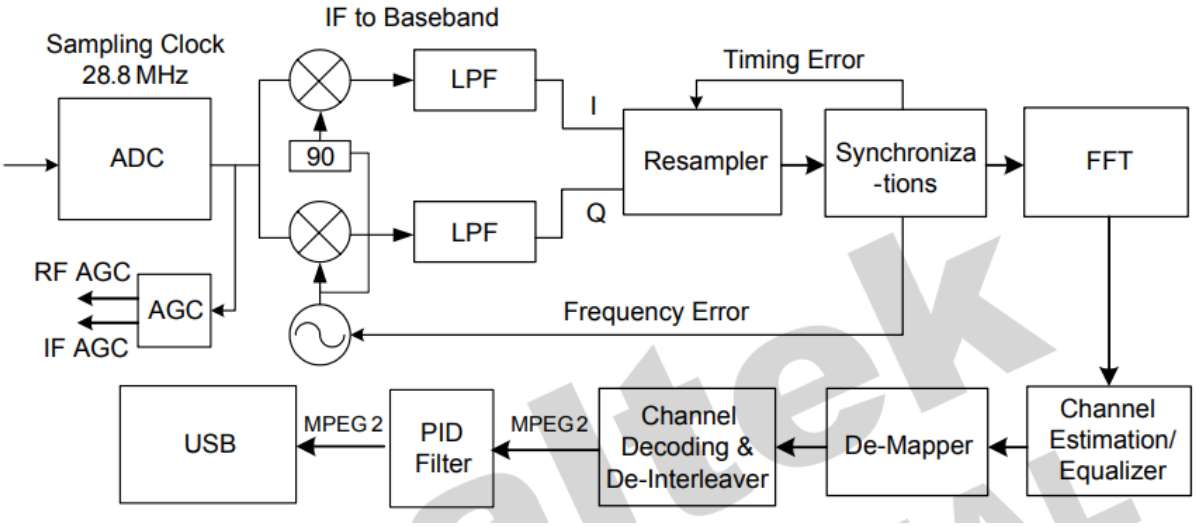


Figure 32: Block diagram of the RTL-SDR. From [40]

A.2 Figma GUI

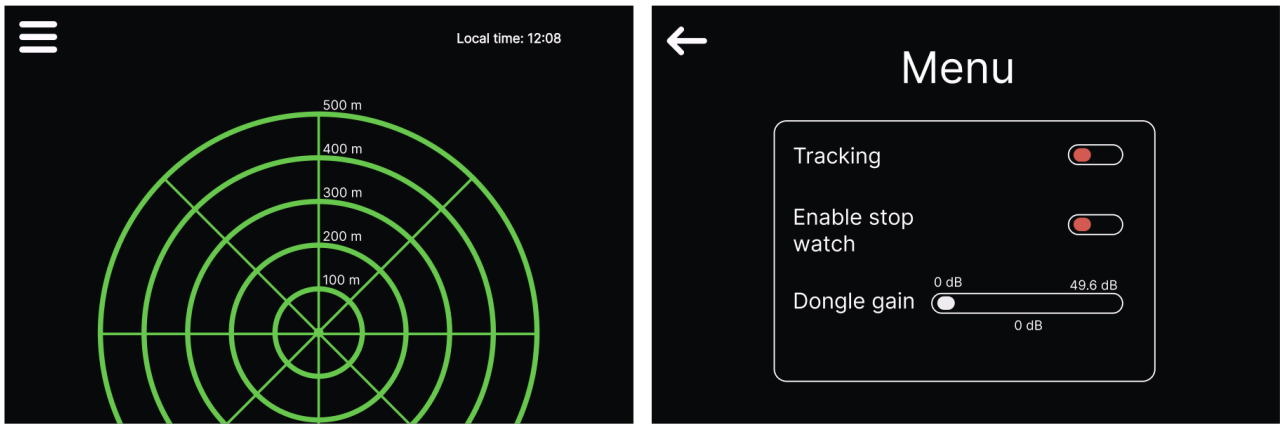


Figure 33: GUI menu when the menu options are turned off.

A.3 UML class diagram

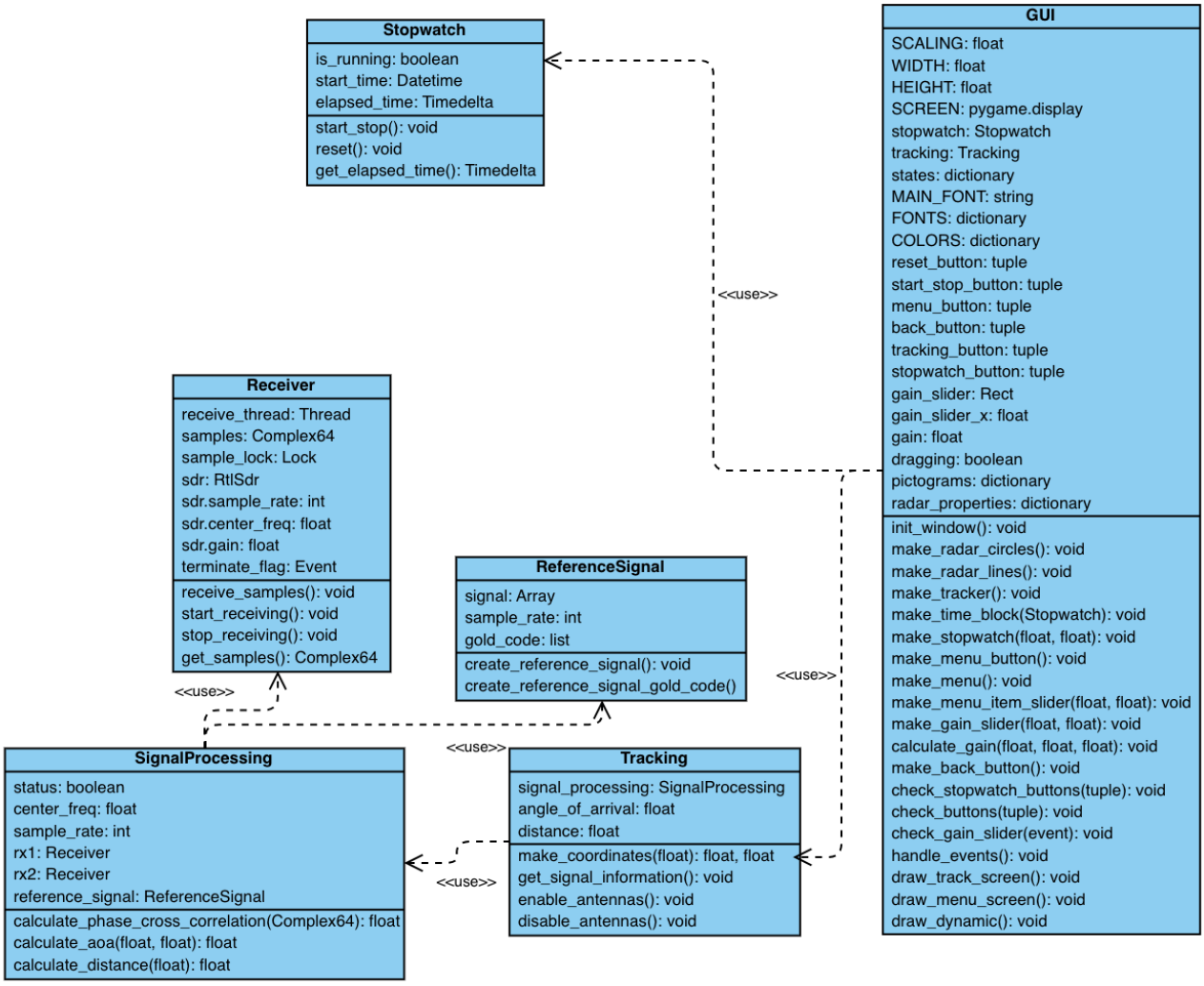


Figure 34: Visual representation of the class structure of the software using Unified Modeling Language

B Application code

The github repository for all of the main code and test code can be found using the following link: https://github.com/Mika-08/DirectionFinding

B.1 Main

```
import pygame
import GUI


def main():
    """
    Main function of the program
    :return: Nothing
    """

    # For running on Mac scaling = 1.4
    scaling = 1.4

    gui = GUI.GUI(scaling)
    gui.init_window()
    pygame.display.update()

    if not gui.states["dummy_mode"]:
        gui.tracking.enable_antennas()

    running = True
    while running:
        gui.draw_dynamic()
        pygame.display.update()
        running = gui.handle_events()

    pygame.quit()


if __name__ == "__main__":
    main()
```

B.2 GUI

```
import pygame
import numpy as np
from datetime import datetime

import Stopwatch
import Tracking


class GUI:
    def __init__(self, scaling):
        """
        Constructor function for GUI class
```

```
        """
        self.SCALING = scaling
        self.WIDTH = 1400 / self.SCALING
        self.HEIGHT = 2 * self.WIDTH / 3
        self.SCREEN = None

        self.states = {
            "stopwatch_is_enabled": True,
            "tracking_is_enabled": True,
            "menu": False,
            # Make true if not antennas are connected
            "dummy_mode": False
        }

        self.stopwatch = Stopwatch.Stopwatch()

        if not self.states["dummy_mode"]:
            self.tracking = Tracking.Tracking()

        pygame.font.init()
        self.MAIN_FONT = 'freesansbold.ttf'
        self.FONTS = {
            "radar_font": pygame.font.Font(self.MAIN_FONT, int(30 / self.SCALING)),
            "time_font": pygame.font.Font(self.MAIN_FONT, int(30 / self.SCALING)),
            "stopwatch_font": pygame.font.Font(self.MAIN_FONT, int(30 /
            ↪  self.SCALING)),
            "menu_font": pygame.font.Font(self.MAIN_FONT, int(100 / self.SCALING)),
            "menu_item_font": pygame.font.Font(self.MAIN_FONT, int(50 /
            ↪  self.SCALING)),
            "gain_slider_font": pygame.font.Font(self.MAIN_FONT, int(30 /
            ↪  self.SCALING))
        }

        self.COLORS = {
            "black": (8, 9, 10),
            "green": (50, 205, 50),
            "red": (240, 84, 79),
            "white": (255, 250, 255)
        }

        # Buttons
        self.reset_button = None
        self.start_stop_button = None
        self.menu_button = None
        self.back_button = None
        self.tracking_button = None
        self.stopwatch_button = None
        self.gain_slider = None

        self.gain_slider_x = 0
        self.gain = None
        self.dragging = None
```

```
        # Pictograms
        self.pictograms = {
            "menu_image": pygame.image.load("images/menu_white.PNG"),
            "back_image": pygame.image.load("images/back_arrow_white.png")
        }

        # Radar properties
        self.radar_properties = {
            "radius": 500 / self.SCALING,
            "circle_center_x": int(self.WIDTH / 2),
            "circle_center_y": int(4 * self.HEIGHT / 5),
        }
        self.radar_properties["circle_center"] =
        ↪  (self.radar_properties["circle_center_x"],
        ↪  self.radar_properties["circle_center_y"])

    def init_window(self):
        """
        Function for initializing the screen
        :return: Nothing
        """
        pygame.init()
        self.SCREEN = pygame.display.set_mode((self.WIDTH, self.HEIGHT))
        pygame.display.set_caption("Hornet tracker")

    def make_radar_circles(self):
        """
        Draw radar circles on the GUI.
        """
        for i in range(5):
            radius_outer_circle = self.radar_properties["radius"] - i * (100 /
            ↪  self.SCALING) + 8.4 / self.SCALING
            pygame.draw.circle(self.SCREEN, self.COLORS['green'],
            ↪  self.radar_properties["circle_center"],  radius_outer_circle)
            pygame.draw.circle(self.SCREEN, self.COLORS['black'],
            ↪  self.radar_properties["circle_center"], radius_outer_circle - 16.8 /
            ↪  self.SCALING)

            text_surface = self.FONTS['radar_font'].render(f"{500 - i * 100}m",
            ↪  True, self.COLORS['white'])
            text_rect = text_surface.get_rect()
            text_rect.center = (self.radar_properties["circle_center"][0] + 49 /
            ↪  self.SCALING, self.radar_properties["circle_center"][1] -
            ↪  radius_outer_circle - 14 / self.SCALING)
            self.SCREEN.blit(text_surface, text_rect)

        pygame.draw.circle(self.SCREEN, self.COLORS['green'],
        ↪  self.radar_properties["circle_center"], 10 / self.SCALING)

    def make_radar_lines(self):
        """
        Draw radar lines on the GUI.
        """
```

```
        for i in range(8):
            angle_rad = np.radians(45 * i)
            end_point_x = self.radar_properties["circle_center"][0] +
            ↪  np.sin(angle_rad) * self.radar_properties["radius"]
            end_point_y = self.radar_properties["circle_center"][1] +
            ↪  np.cos(angle_rad) * self.radar_properties["radius"]
            pygame.draw.line(self.SCREEN, self.COLORS['green'],
            ↪  self.radar_properties["circle_center"], (int(end_point_x),
            ↪  int(end_point_y)), width=int(8.4 / self.SCALING))

    def make_tracker(self):
        """
        Funtion to draw the tracker location on the screen
        :return: Nothing
        """
        if not self.states["dummy_mode"]:
            tracker_x_pos, tracker_y_pos =
            ↪  self.tracking.make_coordinates(self.SCALING)
            tracker_center = (int(tracker_x_pos +
            ↪  self.radar_properties["circle_center_x"]), int(tracker_y_pos +
            ↪  self.radar_properties["circle_center_y"]))
            tracker_radius = 25 / self.SCALING
            pygame.draw.circle(self.SCREEN, self.COLORS['red'], tracker_center,
            ↪  tracker_radius)

    def make_time_block(self, stopwatch):
        """
        Function for making the time block at the right top of the screen
        :param stopwatch: true, then stopwatch is enabled
        :return: Nothing
        """
        rectangle_x = self.WIDTH - 390 / self.SCALING
        rectangle_y = 30 / self.SCALING

        # Add the current time to the screen
        current_time = datetime.now().strftime('%H:%M')
        text_time_surface = self.FONTS.get('time_font').render(f"Local time:
        ↪  {current_time}", True, self.COLORS.get('white'))
        text_time_rect = text_time_surface.get_rect()
        text_time_rect.topleft = (rectangle_x + 45 / self.SCALING, rectangle_y + 40
        ↪  / self.SCALING)

        # Add to the screen
        if not self.states["menu"]:
            self.SCREEN.blit(text_time_surface, text_time_rect)

        # Add the outline and the stopwatch function when the stopwatch is enabled
        if stopwatch:
            self.make_stopwatch(rectangle_x, rectangle_y)

    def make_stopwatch(self, rectangle_x, rectangle_y):
        """
        Method to call if the stopwatch feature is turned on
```

```
    :return: Nothing
    """

    # Rectangle outline
    rect = pygame.Rect(rectangle_x, rectangle_y, 370 / self.SCALING, 260 /
    ↪  self.SCALING)

    # Stopwatch time
    stopwatch_time_str = str(self.stopwatch.get_elapsed_time()).split('.')[0]
    stopwatch_text_surface =
    ↪  self.FONTS.get('stopwatch_font').render(f"Stopwatch:
    ↪  {stopwatch_time_str}", True, self.COLORS.get('white'))
    stopwatch_text_rect = stopwatch_text_surface.get_rect()
    stopwatch_text_rect.topleft = (rectangle_x + 45 / self.SCALING, rectangle_y
    ↪  + 100 / self.SCALING)

    # Stopwatch buttons
    reset_text = self.FONTS.get('stopwatch_font').render("Reset", True,
    ↪  self.COLORS.get('white'))

    # Change the start stop button text dynamically
    start_stop_button_text = "Start"
    start_stop_color = self.COLORS.get('green')
    if self.stopwatch.is_running:
        start_stop_button_text = "Stop"
        start_stop_color = self.COLORS.get('red')

    start_stop_text =
    ↪  self.FONTS.get('stopwatch_font').render(f"{start_stop_button_text}",
    ↪  True, self.COLORS.get('white'))

    reset_rect = reset_text.get_rect()
    reset_rect.center = (rectangle_x + (45 + 60) / self.SCALING, rectangle_y +
    ↪  200 / self.SCALING)
    start_stop_rect = start_stop_text.get_rect()
    start_stop_rect.center = (rectangle_x + (45 + 230) / self.SCALING,
    ↪  rectangle_y + 200 / self.SCALING)

    # Make outlines for the buttons
    x_pos_reset = reset_rect[0] - 35 / self.SCALING
    y_pos_reset = reset_rect[1] - 21 / self.SCALING
    x_pos_start_stop = start_stop_rect[0] - 42 / self.SCALING
    y_pos_start_stop = start_stop_rect[1] - 21 / self.SCALING

    self.reset_button = (x_pos_reset, y_pos_reset, 160 / self.SCALING, 75 /
    ↪  self.SCALING)
    self.start_stop_button = (x_pos_start_stop, y_pos_start_stop, 160 /
    ↪  self.SCALING, 75 / self.SCALING)

    if not self.states["menu"]:
        # Add to the screen
        # Draw the outline rectangle
        pygame.draw.rect(self.SCREEN, self.COLORS.get('white'), rect, width=3,
        ↪  border_radius=30)
```

```
        # Draw the stopwatch text
        self.SCREEN.blit(stopwatch_text_surface, stopwatch_text_rect)

        # Fill the inside
        pygame.draw.rect(self.SCREEN, start_stop_color, self.start_stop_button,
        ↪  border_radius=20)

        # Draw the outlines
        pygame.draw.rect(self.SCREEN, self.COLORS.get('white'),
        ↪  self.reset_button, width=3, border_radius=20)
        pygame.draw.rect(self.SCREEN, self.COLORS.get('white'),
        ↪  self.start_stop_button, width=3, border_radius=20)

        # Draw the button text
        self.SCREEN.blit(reset_text, reset_rect)
        self.SCREEN.blit(start_stop_text, start_stop_rect)

    def make_menu_button(self):
        """
        Function to add the menu icon to the screen
        :return: Nothing
        """
        menu_image_scaled = pygame.transform.scale(self.pictograms["menu_image"],
        ↪  (55, 50))
        menu_rect = menu_image_scaled.get_rect(topleft=(48 / self.SCALING, 30 /
        ↪  self.SCALING))
        self.menu_button = menu_rect

        if not self.states["menu"]:
            self.SCREEN.blit(menu_image_scaled, menu_rect)

    def make_menu(self):
        """
        Function to make the menu
        :return: Nothing
        """
        rectangle_x = 260 / self.SCALING
        rectangle_y = 250 / self.SCALING

        text_menu_surface = self.FONTS.get('menu_font').render("Menu", True,
        ↪  self.COLORS.get('white'))
        text_menu_rect = text_menu_surface.get_rect()
        text_menu_rect.center = (self.WIDTH / 2, 180 / self.SCALING)

        # Make rectangle
        rect = pygame.Rect(rectangle_x, rectangle_y, 875 / self.SCALING, 600 /
        ↪  self.SCALING)

        # Add text
        # Tracking
        text_tracking_surface = self.FONTS.get('menu_item_font').render("Tracking",
        ↪  True, self.COLORS.get('white'))
```

```
text_tracking_rect = text_tracking_surface.get_rect()
text_tracking_rect.topleft = (rectangle_x + 50 / self.SCALING, rectangle_y +
↪  50 / self.SCALING)

# Stopwatch
text_stopwatch_surface =
↪  self.FONTS.get('menu_item_font').render("Enable\nstopwatch", True,
↪  self.COLORS.get('white'))
text_stopwatch_rect = text_stopwatch_surface.get_rect()
text_stopwatch_rect.topleft = (rectangle_x + 50 / self.SCALING, rectangle_y
↪  + 200 / self.SCALING)

# Gain text
text_gain_surface = self.FONTS.get('menu_item_font').render("Dongle gain",
↪  True, self.COLORS.get('white'))
text_gain_rect = text_gain_surface.get_rect()
text_gain_rect.topleft = (rectangle_x + 50 / self.SCALING, rectangle_y + 400
↪  / self.SCALING)

# Gain left bound
text_gain_left_surface = self.FONTS.get('gain_slider_font').render("0 dB",
↪  True, self.COLORS.get('white'))
text_gain_left_rect = text_gain_left_surface.get_rect()
text_gain_left_rect.topleft = (rectangle_x + 380 / self.SCALING, rectangle_y
↪  + 350 / self.SCALING)

# Gain right bound
text_gain_right_surface = self.FONTS.get('gain_slider_font').render("49.6
↪  dB", True, self.COLORS.get('white'))
text_gain_right_rect = text_gain_right_surface.get_rect()
text_gain_right_rect.topleft = (rectangle_x + 675 / self.SCALING,
↪  rectangle_y + 350 / self.SCALING)

# Gain value
text_gain_value_surface =
↪  self.FONTS.get('gain_slider_font').render(f"{self.gain} dB", True,
↪  self.COLORS.get('white'))
text_gain_value_rect = text_gain_value_surface.get_rect()
text_gain_value_rect.topleft = (rectangle_x + 525 / self.SCALING,
↪  rectangle_y + 450 / self.SCALING)

# Make the menu sliders
self.make_menu_item_slider(rectangle_x, rectangle_y)

if self.states["menu"]:
    # Add to the screen
    pygame.draw.rect(self.SCREEN, self.COLORS.get('white'), rect, width=3,
    ↪  border_radius=30)

    self.SCREEN.blit(text_menu_surface, text_menu_rect)
    self.SCREEN.blit(text_tracking_surface, text_tracking_rect)
    self.SCREEN.blit(text_stopwatch_surface, text_stopwatch_rect)
    self.SCREEN.blit(text_gain_surface, text_gain_rect)
```

```
        self.SCREEN.blit(text_gain_left_surface, text_gain_left_rect)
        self.SCREEN.blit(text_gain_right_surface, text_gain_right_rect)
        self.SCREEN.blit(text_gain_value_surface, text_gain_value_rect)

def make_menu_item_slider(self, rectangle_x, rectangle_y):
    """
    Function for making the menu sliders
    :param rectangle_x: x position of the border
    :param rectangle_y: y position of the border
    :return: Nothing
    """

    # TODO: Divide into multiple functions

    rect_tracking = pygame.Rect(rectangle_x + 670 / self.SCALING, rectangle_y +
    ↪  50 / self.SCALING, 120 / self.SCALING, 45 / self.SCALING)

    self.tracking_button = rect_tracking

    rect_stopwatch = pygame.Rect(rectangle_x + 670 / self.SCALING, rectangle_y +
    ↪  200 / self.SCALING, 120 / self.SCALING, 45 / self.SCALING)

    self.stopwatch_button = rect_stopwatch

    tracking_dot_x_pos = 371 / self.SCALING
    tracking_dot_color = self.COLORS.get('red')
    if self.states["tracking_is_enabled"]:
        tracking_dot_x_pos = 420 / self.SCALING
        tracking_dot_color = self.COLORS.get('green')

    rect_tracking_dot = pygame.Rect(tracking_dot_x_pos + 575 / self.SCALING,
    ↪  rectangle_y + 57 / self.SCALING, 40 / self.SCALING, 32 / self.SCALING)

    stopwatch_dot_x_pos = 371 / self.SCALING
    stopwatch_dot_color = self.COLORS.get('red')
    if self.states["stopwatch_is_enabled"]:
        stopwatch_dot_x_pos = 420 / self.SCALING
        stopwatch_dot_color = self.COLORS.get('green')

    rect_stopwatch_dot = pygame.Rect(stopwatch_dot_x_pos + 575 / self.SCALING,
    ↪  rectangle_y + 207 / self.SCALING, 40 / self.SCALING, 32 / self.SCALING)

    self.make_gain_slider(rectangle_x, rectangle_y)

    if self.states["menu"]:
        pygame.draw.rect(self.SCREEN, self.COLORS.get('white'), rect_tracking,
        ↪  width=3, border_radius=20)
        pygame.draw.rect(self.SCREEN, self.COLORS.get('white'), rect_stopwatch,
        ↪  width=3, border_radius=20)
        pygame.draw.rect(self.SCREEN, tracking_dot_color, rect_tracking_dot,
        ↪  border_radius=20)
        pygame.draw.rect(self.SCREEN, stopwatch_dot_color, rect_stopwatch_dot,
        ↪  border_radius=20)
```

```
def make_gain_slider(self, rectangle_x, rectangle_y):
    """
    Make the gain slider
    :param rectangle_x: x position of the menu rectangle
    :param rectangle_y: y position of the menu rectangle
    :return: Nothing
    """

    slider_x = rectangle_x + 370 / self.SCALING
    slider_y = rectangle_y + 400 / self.SCALING

    slider_width = 420 / self.SCALING
    dot_width = 40 / self.SCALING

    rect_gain = pygame.Rect(slider_x, slider_y, slider_width, 45 / self.SCALING)

    # self.gain_slider_x = slider_x

    if self.gain_slider_x < slider_x:
        self.gain_slider_x = slider_x

    if self.gain_slider_x > slider_x + 261.42857142857144:
        self.gain_slider_x = slider_x + 261.42857142857144

    rect_gain_dot = pygame.Rect(self.gain_slider_x + 7 / self.SCALING, slider_y
    ↪  + 7 / self.SCALING, dot_width, 32 / self.SCALING)

    self.gain_slider = rect_gain_dot

    self.calculate_gain(slider_x, slider_width, dot_width)

    if self.states["menu"]:
        pygame.draw.rect(self.SCREEN, self.COLORS.get('white'), rect_gain,
        ↪  width=3, border_radius=20)
        pygame.draw.rect(self.SCREEN, self.COLORS.get('white'), rect_gain_dot,
        ↪  border_radius=20)

def calculate_gain(self, slider_x, slider_width, dot_width):
    """
    Calculate the gain value
    :param slider_x: The x position of the slider rectangle
    :param slider_width: The width of the slider rectangle
    :param dot_width: The width of the slider dot
    :return: Nothing
    """
    gain_values = [0.0, 0.9, 1.4, 2.7, 3.7, 7.7, 8.7, 12.5, 14.4, 15.7, 16.6,
    ↪  19.7, 20.7, 22.9, 25.4, 28.0, 29.7, 32.8, 33.8, 36.4, 37.2, 38.6, 40.2,
    ↪  42.1, 43.4, 43.9, 44.5, 48.0, 49.6]

    min_value_gain = min(gain_values)
    max_value_gain = max(gain_values)
```

```
        normalized_gain_values = [(value - min_value_gain) / (max_value_gain -
        ↪  min_value_gain) * (slider_width - 14 / self.SCALING - dot_width) for
        ↪  value in gain_values]

        slider_value = (self.gain_slider_x - slider_x) / (slider_width - 14 /
        ↪  self.SCALING - dot_width) * \ (max_value_gain - min_value_gain) +
        ↪  min_value_gain

        self.gain = min(gain_values, key=lambda x: abs(x - slider_value))

    def make_back_button(self):
        """
        Make the back button
        :return: Nothing
        """
        back_image_scaled = pygame.transform.scale(self.pictograms["back_image"],
        ↪  (60, 50))
        back_rect = back_image_scaled.get_rect(topleft=(48 / self.SCALING, 30 /
        ↪  self.SCALING))
        self.back_button = back_rect

        if self.states["menu"]:
            self.SCREEN.blit(back_image_scaled, back_rect)

    def check_stopwatch_buttons(self, mouse_pos):
        """
        Check is the stopwatch buttons are clicked
        :param mouse_pos: position of the mouse
        :return: Nothing
        """
        # Make the hit boxes for the stop watch buttons
        reset_box = pygame.Rect(self.reset_button)
        start_stop_box = pygame.Rect(self.start_stop_button)

        if reset_box.collidepoint(mouse_pos):
            self.stopwatch.reset()
        elif start_stop_box.collidepoint(mouse_pos):
            self.stopwatch.start_stop()

    def check_buttons(self, mouse_pos):
        """
        Function for checking if a button is clicked
        :param mouse_pos: position of the mouse
        :return: Nothing
        """

        if self.states["menu"]:
            if self.back_button.collidepoint(mouse_pos):
                self.states["menu"] = False

            if self.tracking_button.collidepoint(mouse_pos) and
            ↪  self.states["tracking_is_enabled"]:
                self.states["tracking_is_enabled"] = False
```

```
                if not self.states["dummy_mode"]:
                    self.tracking.disable_antennas()

            elif self.tracking_button.collidepoint(mouse_pos) and not
            ↪  self.states["tracking_is_enabled"]:
                self.states["tracking_is_enabled"] = True
                if not self.states["dummy_mode"]:
                    self.tracking.enable_antennas()

            if self.stopwatch_button.collidepoint(mouse_pos) and
            ↪  self.states["stopwatch_is_enabled"]:
                self.states["stopwatch_is_enabled"] = False

            elif self.stopwatch_button.collidepoint(mouse_pos) and not
            ↪  self.states["stopwatch_is_enabled"]:
                self.states["stopwatch_is_enabled"] = True

        elif not self.states["menu"]:
            if self.menu_button.collidepoint(mouse_pos):
                self.states["menu"] = True

            if self.states["stopwatch_is_enabled"]:
                self.check_stopwatch_buttons(mouse_pos)

    def check_gain_slider(self, event):
        """
        Check if the slider is adjusted
        :param event: Event in the event list
        :return: Noting
        """
        if self.states["menu"]:
            if event.type == pygame.MOUSEBUTTONDOWN:
                if self.gain_slider.collidepoint(event.pos):
                    self.dragging = True
            elif event.type == pygame.MOUSEBUTTONUP:
                self.dragging = False
            elif event.type == pygame.MOUSEMOTION:
                if self.dragging:
                    mouse_x, _ = event.pos
                    self.gain_slider_x = mouse_x

    def handle_events(self):
        """
        Event handler
        :return: Nothing
        """
        for event in pygame.event.get():
            mouse_pos = pygame.mouse.get_pos()
            if event.type == pygame.QUIT:
                if not self.states["dummy_mode"]:
                    self.tracking.disable_antennas()
                return False
            elif event.type == pygame.MOUSEBUTTONDOWN:
```

```
                self.check_buttons(mouse_pos)

            self.check_gain_slider(event)
        return True

    def draw_track_screen(self):
        """
        Draw the tracking screen
        :return: Nothing
        """
        # Make radar
        self.make_radar_circles()
        self.make_radar_lines()

        if self.states["tracking_is_enabled"]:
            self.make_tracker()

        # Make menu button
        self.make_menu_button()
        self.make_back_button()

        # Show or not show the stopwatch
        if self.states["stopwatch_is_enabled"]:
            self.make_time_block(True)
        elif not self.states["stopwatch_is_enabled"]:
            self.make_time_block(False)

    def draw_menu_screen(self):
        """
        Draw the menu screen
        :return: Nothing
        """
        self.make_menu()
        self.make_back_button()

    def draw_dynamic(self):
        """
        Function for dynamically drawing the items on the screen
        :return: Nothing
        """
        self.make_menu_button()
        self.make_back_button()
        self.make_time_block(self.stopwatch)
        self.make_menu()

        self.SCREEN.fill(self.COLORS.get('black'))

        if self.states["menu"]:
            self.SCREEN.fill(self.COLORS.get('black'))

            # print("menu show")
            self.draw_menu_screen()
```

```
        if not self.states["menu"]:
            self.SCREEN.fill(self.COLORS.get('black'))

            # print("tracking show")
            self.draw_track_screen()
```

B.3 Stopwatch

```
from datetime import datetime, timedelta


class Stopwatch:
    def __init__(self):
        """
        Initializer function for the Stopwatch class
        """
        self.is_running = False
        self.start_time = None
        self.elapsed_time = timedelta()

    def start_stop(self):
        """
        Function for starting and stopping the stopwatch
        :return: Nothing
        """
        if self.is_running:
            self.is_running = False
            self.elapsed_time += datetime.now() - self.start_time
        else:
            self.is_running = True
            self.start_time = datetime.now()

    def reset(self):
        """
        Function for resetting the stopwatch
        :return: Nothing
        """
        self.is_running = False
        self.start_time = None
        self.elapsed_time = timedelta()

    def get_elapsed_time(self):
        """
        Get the elapsed time
        :return: The elapsed time
        """
        if self.is_running:
            return self.elapsed_time + (datetime.now() - self.start_time)
        else:
            return self.elapsed_time
```

B.4 Tracking

```
import numpy as np
import SignalProcessing


class Tracking:
    def __init__(self):
        """
        Constructor function for the tracking class
        """
        self.signal_processing = SignalProcessing.SignalProcessing()
        self.angle_of_arrival = None
        self.distance = None

    def enable_antennas(self):
        """
        Function to enable the receiving antennas
        :return: Nothing
        """
        self.signal_processing.enable_receiving()

    def disable_antennas(self):
        """
        Function to disable the receiving antennas
        :return: Nothing
        """
        self.signal_processing.disable_receiving()

    def get_signal_information(self):
        """
        Function for getting the information from the signal processing object
        :return: Nothing
        """
        self.angle_of_arrival = self.signal_processing.calculate_aoa()
        self.distance = self.signal_processing.calculate_distance()

    def make_coordinates(self, scaling):
        """
        Function for making the coordinates of the dot
        :return:
        """
        self.get_signal_information()

        ring_number = (self.distance + 99) // 100
        if ring_number > 5:
            ring_number = 5

        line_length = ring_number * 100 / scaling  # Think of solution

        # - 90 to let the top correspond with 0 degrees
        angle_radians = np.radians(self.angle_of_arrival - 90)
        x_pos = np.cos(angle_radians) * line_length
```

```
        y_pos = np.sin(angle_radians) * line_length

        return x_pos, y_pos
```

B.5 Receiver

```
import numpy as np
import threading
from rtlsdr import RtlSdr


class Receiver:
    def __init__(self, center_freq, sample_rate, gain='auto', device_index=0):
        """
        Constructor for the ReceiverTest class
        :param center_freq: center frequency of the signal
        :param sample_rate: sampling rate
        :param gain: gain of the dongle
        :param device_index: index of the RTL-SDR dongle
        """
        self.receive_thread = None
        self.samples = np.zeros(int(5 * sample_rate), dtype=np.complex64)
        self.sample_lock = threading.Lock()
        self.sdr = RtlSdr(device_index=device_index)

        # Configure SDR settings
        self.sdr.sample_rate = sample_rate  # Hz
        self.sdr.center_freq = center_freq  # Hz
        self.sdr.gain = gain

        # Flag to stop the receiving thread
        self.terminate_flag = threading.Event()

    def receive_samples(self):
        """
        Function to receive the samples
        :return: Nothing
        """
        while not self.terminate_flag.is_set():
            samples = self.sdr.read_samples(256 * 1024)  # Read samples from the
            ↪  SDR
            with self.sample_lock:
                self.samples = np.roll(self.samples, -len(samples))
                self.samples[-len(samples):] = samples

    def start_receiving(self):
        """
        Function to make a thread to receive the samples
        :return: Nothing
        """
        self.receive_thread = threading.Thread(target=self.receive_samples)
        self.receive_thread.daemon = True  # Allows the program to exit even if the
        ↪  thread is running
```

```
        self.receive_thread.start()

    def stop_receiving(self):
        """
        Function to stop receiving when the terminate_flag has been set
        :return: Nothing
        """
        self.terminate_flag.set()
        self.receive_thread.join()
        self.sdr.close()

    def get_samples(self):
        """
        Function to get the samples using a data lock
        :return: The samples
        """
        with self.sample_lock:
            return self.samples
```

B.6 SignalProcessing

```
import Kalman
import numpy as np
import Receiver
import ReferenceSignal
import scipy


def zero_pad_array(arr, desired_length):
    """
    Function that adds additional zeros to the array
    :param arr: Array to be altered
    :param desired_length: Length which the altered array needs to be
    :return: The altered array
    """
    current_length = len(arr)
    if current_length < desired_length:
        padding = desired_length - current_length
        arr = np.pad(arr, (0, padding), 'constant', constant_values=(0,))
    return arr


def time_delay(ref, samples1, samples2, sample_rate):
    """
    Function to calculate the time difference and phase difference between two
    ↪  signal
    :param ref: Reference signal
    :param samples1: Received signal 1
    :param samples2: Received signal 2
    :return: The time difference and phase difference
    """

    downsampling_factor = 1
```

```
    ref = ref[::downsampling_factor]
    samples1 = samples1[::downsampling_factor]
    samples2 = samples2[::downsampling_factor]

    length_ref = len(ref)
    length_samples = len(samples1)

    total_length = length_ref + length_samples - 1

    ref = zero_pad_array(ref, total_length)
    samples1 = zero_pad_array(samples1, total_length)
    samples2 = zero_pad_array(samples2, total_length)

    mult1 = np.flip(scipy.fft.fft(np.conj(ref))) * scipy.fft.fft(samples1)
    mult2 = np.flip(scipy.fft.fft(np.conj(ref))) * scipy.fft.fft(samples2)

    if np.argmax(np.abs(scipy.fft.ifft(mult2))) >
    ↪  np.argmax(np.abs(scipy.fft.ifft(mult1))):
        ifft = np.abs(scipy.fft.ifft(mult2 / mult1))

    else:
        ifft = np.abs(scipy.fft.ifft(mult1 / mult2))

    td = np.linspace(0, total_length / sample_rate, total_length)
    est_delay = (td[np.argmax(np.abs(ifft))] * downsampling_factor) % 1

    T = 1
    phase_delay = 2 * np.pi * 1 / T * est_delay
    phase_delay_degrees = np.rad2deg(phase_delay)

    return est_delay, phase_delay


class SignalProcessing:
    def __init__(self):
        """
        Constructor for the signal processing class
        """
        self.status = False
        self.kalman = Kalman.Kalman()
        self.center_freq = 434e6
        self.sample_rate = 2.048e6

        self.rx1 = Receiver.Receiver(self.center_freq, self.sample_rate,
        ↪  device_index=0)
        self.rx2 = Receiver.Receiver(self.center_freq, self.sample_rate,
        ↪  device_index=1)
        self.reference_signal = ReferenceSignal.ReferenceSignal(self.sample_rate)

        self.past_data = []

    def disable_receiving(self):
        """
```

```
        Function to let the sdr dongles receive
        :return: Nothing
        """
        self.rx1.stop_receiving()
        self.rx2.stop_receiving()

        self.save_data()
        self.past_data = []

    def enable_receiving(self):
        """
        Function to let the sdr dongles stop with receiving
        :return: Nothing
        """
        self.rx1.start_receiving()
        self.rx2.start_receiving()

    def save_data(self):
        """
        Function to save the last tracking to a file
        :return: Nothing
        """
        print(self.past_data)
        output_file = open("Output.txt", "w")
        for data in self.past_data:
            output_file.write(str(data))
            output_file.write("\n")

    def calculate_delta_phi(self):
        """
        Function to calculate the phase difference between the two received signals
        :return: delta_phi, the phase difference
        """
        # Get samples
        samples1 = self.rx1.get_samples()
        samples2 = self.rx2.get_samples()

        modulated_ref = self.reference_signal.signal

        delay, phase = time_delay(modulated_ref, samples1, samples2,
        ↪  self.sample_rate)
        delta_phi = phase

        return delta_phi

    def calculate_aoa(self):
        """
        Function to calculate the angle of arrival with the delta_phi
        :return: Angle of arrival
        """
        delta_phi = self.calculate_delta_phi()
        # Ensure the phase difference is within the range [-pi, pi]
        delta_phi = np.arctan2(np.sin(delta_phi), np.cos(delta_phi))
```

```
        # Calculate the angle of arrival
        sin_theta = delta_phi / np.pi
        # Ensure sin_theta is in the valid range [-1, 1]
        sin_theta = np.clip(sin_theta, -1, 1)
        angle_of_arrival = np.arcsin(sin_theta)
        # Convert to degrees
        angle_of_arrival_degrees = np.rad2deg(angle_of_arrival)

        print(angle_of_arrival_degrees)

        self.past_data.append(angle_of_arrival_degrees)
        return angle_of_arrival_degrees


    def calculate_distance(self):
        """
        Function to calculate the distance to the receiver using the RSSI
        Not implemented
        :return: Distance
        """
        distance = 250

        return distance
```

B.7 ReferenceSignal

```
import numpy as np

def makeSignal(ref, sample_rate):
    # Define the carrier frequency
    carrier_frequency = 434e6

    # Generate the time vector for the base signal
    t = np.arange(len(ref)) / sample_rate

    # Generate the carrier signal
    carrier_signal = np.cos(2 * np.pi * carrier_frequency * t)

    # Modulate the base signal with the carrier
    modulated_signal = ref * carrier_signal

    return modulated_signal

class ReferenceSignal:
    def __init__(self, sample_rate):
        """
        Constructor function for the reference signal class
        :param sample_rate: Sample rate with which the dongles sample
        """
        self.signal = None
```

```
        self.sample_rate = sample_rate
        self.gold_code = [1, 1, 1, 0, 0, 1, 1, 1, 0, 1, 0, 1, 1, 0, 0, 1, 0, 0, 1,
        ↪  0, 0, 1, 1, 1, 0, 0, 1, 0, 0, 1, 0]
        self.create_reference_signal()

    def create_reference_signal_gold_code(self):
        signal = []
        bit_time = 0.01613

        bit_duration = int(bit_time * self.sample_rate)

        for value in self.gold_code:
            if value == 1:
                signal.append(np.ones(bit_duration))
            else:
                signal.append(np.zeros(bit_duration))

        # Concatenate the list of arrays into a single array
        self.signal = np.concatenate(signal)

    def create_reference_signal(self):
        """
        Function that creates a reference signal form the code
        :return: Nothing
        """
        on_duration = int(0.03 * self.sample_rate)
        off_duration_short = int(0.03 * self.sample_rate)
        off_duration_long = int(0.91 * self.sample_rate)

        reference_signal = np.concatenate([
            np.ones(on_duration),
            np.zeros(off_duration_short),
            np.ones(on_duration),
            np.zeros(off_duration_long)
        ])

        modulated_ref = makeSignal(reference_signal, self.sample_rate)

        self.signal = modulated_ref
```

C Test code

C.1 Receive signal

```
import numpy as np
import matplotlib.pyplot as plt
import Receiver
from matplotlib.animation import FuncAnimation

center_freq = 434e6
sample_rate = 2.048e6
receiver = Receiver.Receiver(center_freq, sample_rate)
receiver.start_receiving()
duration = 5  # Duration in seconds


# Create a figure and axis for plotting
fig, (ax1) = plt.subplots(1, 1, figsize=(10, 8))

x_data = np.linspace(-duration, 0, int(duration * sample_rate))
y_data1 = np.zeros(int(duration * sample_rate))
phase_data = []

line1, = ax1.plot(x_data, y_data1, lw=2, label='Receiver 1')

ax1.set_ylim(0, 2)
ax1.set_xlabel('Time (s)')
ax1.set_ylabel('Amplitude')
ax1.set_title('Received Signal - RTL-SDR 1')
ax1.legend(loc='upper right')


def init():
    """
    Initialize the plot
    :return:
    """
    line1.set_ydata(np.zeros_like(x_data))
    return line1,


def update(frame):
    """
    Function that updates the plot real-time
    :return: The lines to plot
    """
    samples1 = receiver.get_samples()

    if samples1 is not None:
        # Update the plot data for the received signal
        y_data1[:] = np.abs(samples1[:int(duration * sample_rate)])
        line1.set_ydata(y_data1)

    return line1,
```

```
# Set up the animation
ani = FuncAnimation(fig, update, init_func=init, blit=True, interval=50,
↪  cache_frame_data=False)

# Show the plot
plt.tight_layout()
plt.show()

receiver.stop_receiving()
```

C.2 Receive signals using two antennas

```
import numpy as np
import threading
import matplotlib.pyplot as plt
from matplotlib.animation import FuncAnimation
from rtlsdr import RtlSdr
import Receiver

# Initialize receivers with the center frequency and sample rate
center_freq = 434e6
sample_rate = 2.048e6

receiver1 = Receiver.Receiver(center_freq, sample_rate, device_index=0)
receiver2 = Receiver.Receiver(center_freq, sample_rate, device_index=1)

receiver1.start_receiving()
receiver2.start_receiving()

# Create a figure and axis for plotting
fig, (ax1, ax2) = plt.subplots(2, 1, figsize=(10, 6))
x_data = np.linspace(-5, 0, int(5 * sample_rate))
y_data1 = np.zeros(int(5 * sample_rate))
y_data2 = np.zeros(int(5 * sample_rate))

line1, = ax1.plot(x_data, y_data1, lw=2, label='Receiver 1')
line2, = ax2.plot(x_data, y_data2, lw=2, label='Receiver 2')

ax1.set_ylim(0, 2)
ax2.set_ylim(0, 2)
ax1.set_xlabel('Time (s)')
ax2.set_xlabel('Time (s)')
ax1.set_ylabel('Amplitude')
ax2.set_ylabel('Amplitude')
ax1.set_title('Received Signal - RTL-SDR 1')
ax2.set_title('Received Signal - RTL-SDR 2')

ax1.legend(loc='upper right')
ax2.legend(loc='upper right')
```

```
def init():
    """
    Initialize the plot
    :return:
    """
    line1.set_ydata(np.zeros_like(y_data1))
    line2.set_ydata(np.zeros_like(y_data2))
    return line1, line2


def update(frame):
    """
    Function that updates the plot real-time
    :return: The lines to plot
    """
    samples1 = receiver1.get_samples()
    samples2 = receiver2.get_samples()

    if samples1 is not None and samples2 is not None:
        # Update the plot data
        y_data1[:] = np.abs(samples1[:int(5 * sample_rate)])
        y_data2[:] = np.abs(samples2[:int(5 * sample_rate)])

        line1.set_ydata(y_data1)
        line2.set_ydata(y_data2)

    return line1, line2


# Set up the animation
ani = FuncAnimation(fig, update, init_func=init, blit=True, interval=50,
↪  cache_frame_data=False)

# Show the plot
plt.tight_layout()
plt.show()

# Stop receiving samples when closing the plot
receiver1.stop_receiving()
receiver2.stop_receiving()
```

C.3 Phase difference calculation

```
import numpy as np
import scipy
import matplotlib.pyplot as plt

center_freq = 434e6
sample_rate = 2.048e6
duration = 5  # Duration in seconds
wavelength = 3e8 / center_freq  # Speed of light divided by center frequency
```

```
on_duration = int(0.03 * sample_rate)
off_duration_short = int(0.03 * sample_rate)
off_duration_long = int(0.91 * sample_rate)

ref = np.concatenate([
    np.ones(on_duration),
    np.zeros(off_duration_short),
    np.ones(on_duration),
    np.zeros(off_duration_long)
])


def makeSignal(ref, sample_rate):
    # Define the carrier frequency
    carrier_frequency = 434e6

    # Generate the time vector for the base signal
    t = np.arange(len(ref)) / sample_rate

    # Generate the carrier signal
    carrier_signal = np.cos(2 * np.pi * carrier_frequency * t)

    # Modulate the base signal with the carrier
    modulated_signal = ref * carrier_signal

    return modulated_signal


def addTimeShift(s, t0, sample_rate):
    n = len(s)
    fft_signal = scipy.fft.fft(s)

    # Calculate the frequencies for each component
    freqs = np.fft.fftfreq(n, d=1 / sample_rate)

    # Create the phase shift array for the given time shift t0
    phase_shifts = np.exp(-1j * 2 * np.pi * freqs * t0)

    # Apply the phase shifts to the FFT signal
    shifted_fft_signal = fft_signal * phase_shifts

    # Compute the inverse FFT to get the time-shifted signal
    shifted_signal = scipy.fft.ifft(shifted_fft_signal)

    return shifted_signal


def zero_pad_array(arr, desired_length):
    """
    Function that adds additional zeros to the array
    :param arr: Array to be altered
    :param desired_length: Length which the altered array needs to be
```

```
    :return: The altered array
    """
    current_length = len(arr)
    if current_length < desired_length:
        padding = desired_length - current_length
        arr = np.pad(arr, (0, padding), 'constant', constant_values=(0,))
    return arr


def makePeak(ref, samples1):
    """
    Function to calculate the time difference and phase difference between the
    ↪  received signal and the reference signal
    :param ref: Reference signal
    :param samples1: Received signal
    :return: The time delay and phase difference
    """
    downsampling_factor = 1
    ref = ref[::downsampling_factor]
    samples1 = samples1[::downsampling_factor]

    length_ref = len(ref)
    length_samples = len(samples1)

    total_length = length_ref + length_samples - 1

    ref = zero_pad_array(ref, total_length)
    samples1 = zero_pad_array(samples1, total_length)

    mult1 = np.flip(scipy.fft.fft(np.conj(ref))) * scipy.fft.fft(samples1)
    ifft = np.abs(scipy.fft.ifft(mult1))

    td = np.linspace(0, total_length / sample_rate, total_length)
    # print(td)
    est_delay = td[np.argmax(np.abs(ifft))] * downsampling_factor

    T = 1
    phase_delay = 2 * np.pi * 1 / T * est_delay
    phase_delay_degrees = np.rad2deg(phase_delay)

    plt.plot(td, ref, label="Reference signal")
    plt.plot(td, samples1, label="Test signal")
    plt.title(F"Phase difference at 0.05 seconds time delay\n"
              F"Calculated value {phase_delay_degrees:.03f} degrees")
    plt.ylabel("Amplitude")
    plt.xlabel("Time (s)")
    plt.legend()
    plt.show()

    return est_delay, phase_delay


def makeNoise(signal):
```

```
    target_snr_db = 20

    x_watts = signal ** 2
    # Calculate signal power and convert to dB
    sig_avg_watts = np.mean(x_watts)
    sig_avg_db = 10 * np.log10(sig_avg_watts)
    # Calculate noise according to [2] then convert to watts
    noise_avg_db = sig_avg_db - target_snr_db
    noise_avg_watts = 10 ** (noise_avg_db / 10)
    # Generate a sample of white noise
    mean_noise = 0
    noise_volts = np.random.normal(mean_noise, np.sqrt(noise_avg_watts),
    ↪  len(x_watts))

    # Add the noise to the data.
    return signal + noise_volts


modulated_ref = makeSignal(ref, sample_rate)
shifted = addTimeShift(modulated_ref, 0.05, sample_rate)
shifted = makeNoise(shifted)

delay1, phase1 = makePeak(modulated_ref, shifted)
print(F"The time difference between the reference and the shifted signal is:
↪  {delay1} seconds."
      F" The phase difference is: {np.rad2deg(phase1)} degrees.")
```

C.4 Received signal phase difference

```
import numpy as np
import scipy
import matplotlib.pyplot as plt
import Receiver
from matplotlib.animation import FuncAnimation
import time

center_freq = 434e6
sample_rate = 2.048e6
receiver = Receiver.Receiver(center_freq, sample_rate)
receiver.start_receiving()
duration = 5  # Duration in seconds
wavelength = 3e8 / center_freq  # Speed of light divided by center frequency


on_duration = int(0.03 * sample_rate)
off_duration_short = int(0.03 * sample_rate)
off_duration_long = int(0.91 * sample_rate)

# Make the reference signal
ref = np.concatenate([
    np.ones(on_duration),
    np.zeros(off_duration_short),
```

```
    np.ones(on_duration),
    np.zeros(off_duration_long)
])


def makeSignal(ref, sample_rate):
    # Define the carrier frequency
    carrier_frequency = 434e6

    # Generate the time vector for the base signal
    t = np.arange(len(ref)) / sample_rate

    # Generate the carrier signal
    carrier_signal = np.cos(2 * np.pi * carrier_frequency * t)

    # Modulate the base signal with the carrier
    modulated_signal = ref * carrier_signal

    return modulated_signal


def zero_pad_array(arr, desired_length):
    """
    Function that adds additional zeros to the array
    :param arr: Array to be altered
    :param desired_length: Length which the altered array needs to be
    :return: The altered array
    """
    current_length = len(arr)
    if current_length < desired_length:
        padding = desired_length - current_length
        arr = np.pad(arr, (0, padding), 'constant', constant_values=(0,))
    return arr


def makePeak(ref, samples1):
    """
    Function to calculate the time difference and phase difference between the
    ↪  received signal and the reference signal
    :param ref: Reference signal
    :param samples1: Received signal
    :return: The time delay and phase difference
    """
    begin_time = time.time()

    downsampling_factor = 1
    ref = ref[::downsampling_factor]
    samples1 = samples1[::downsampling_factor]

    length_ref = len(ref)
    length_samples = len(samples1)

    total_length = length_ref + length_samples - 1
```

```
    ref = zero_pad_array(ref, total_length)
    samples1 = zero_pad_array(samples1, total_length)

    mult1 = np.flip(scipy.fft.fft(np.conj(ref))) * scipy.fft.fft(samples1)
    ifft = np.abs(scipy.fft.ifft(mult1))

    td = np.linspace(0, total_length / sample_rate, total_length)
    # print(td)
    est_delay = (td[np.argmax(np.abs(ifft))] * downsampling_factor) % 1

    T = 1
    phase_delay = 2 * np.pi * 1 / T * est_delay
    phase_delay_degrees = np.rad2deg(phase_delay)

    time_delta = time.time() - begin_time
    print(F"Elapsed time  is {time_delta} seconds")

    return est_delay, phase_delay


modulated_ref = makeSignal(ref, sample_rate)

# Create a figure and axis for plotting
fig, (ax1, ax2) = plt.subplots(2, 1, figsize=(10, 8))

x_data = np.linspace(-duration, 0, int(duration * sample_rate))
y_data1 = np.zeros(int(duration * sample_rate))
phase_data = []

line1, = ax1.plot(x_data, y_data1, lw=2, label='Receiver 1')
line2, = ax2.plot([], [], lw=2, label='Phase Difference')

ax1.set_ylim(0, 2)
ax1.set_xlabel('Time (s)')
ax1.set_ylabel('Amplitude')
ax1.set_title('Received Signal - RTL-SDR 1')
ax1.legend(loc='upper right')

ax2.set_ylim(0, 360)
ax2.set_xlim(-duration, 0)
ax2.set_xlabel('Time (s)')
ax2.set_ylabel('Phase Difference (degrees)')
ax2.set_title('Phase Difference')
ax2.legend(loc='upper right')


def init():
    """
    Initialize the plot
    :return:
    """
    line1.set_ydata(np.zeros_like(x_data))
```

```
    line2.set_data([], [])
    return line1, line2


def update(frame):
    """
    Function that updates the plot real-time
    :return: The lines to plot
    """
    samples1 = receiver.get_samples()

    if samples1 is not None:
        # Update the plot data for the received signal
        y_data1[:] = np.abs(samples1[:int(duration * sample_rate)])
        line1.set_ydata(y_data1)

        # Calculate phase difference
        time, phase_difference = makePeak(modulated_ref, samples1)
        print(F"The time difference between the reference and the shifted signal is:
        ↪  {time} seconds."
              F" The phase difference is: {np.rad2deg(phase_difference)} degrees.")
        phase_difference_degrees = np.degrees(phase_difference)  # Convert radians
        ↪  to degrees

        # Update the phase difference plot
        phase_data.append(phase_difference_degrees)
        if len(phase_data) > len(x_data):
            phase_data.pop(0)

        line2.set_data(np.linspace(-duration, 0, len(phase_data)), phase_data)

    return line1, line2


# Set up the animation
ani = FuncAnimation(fig, update, init_func=init, blit=True, interval=50,
↪  cache_frame_data=False)

# Show the plot
plt.tight_layout()
plt.show()

receiver.stop_receiving()
```

C.5 Phase difference two signals calculation

```
import numpy as np
import scipy
import matplotlib.pyplot as plt


center_freq = 434e6
```

```
sample_rate = 2.048e6
duration = 5  # Duration in seconds
wavelength = 3e8 / center_freq  # Speed of light divided by center frequency


def plot(signal1, signal2, signal3, phase):
    y1 = np.arange(0, len(signal1))/1000000
    y2 = np.arange(0, len(signal2))/1000000
    y3 = np.arange(0, len(signal3))/1000000

    plt.plot(y1, np.abs(signal1), label="Reference signal")
    plt.plot(y2, np.abs(signal2), label="Shifted signal 1, 0.05 s")
    plt.plot(y3, np.abs(signal3), label="Shifted signal 2, 0.08 s")
    plt.legend()
    plt.title(F"Reference signal and two shifted signals \n"
              F"Calculated phase difference between the two signals: {phase:.01f}
              ↪  degrees.")
    plt.ylabel("Amplitude")
    plt.xlabel("Time (s)")
    plt.show()


on_duration = int(0.03 * sample_rate)
off_duration_short = int(0.03 * sample_rate)
off_duration_long = int(0.91 * sample_rate)

ref = np.concatenate([
    np.ones(on_duration),
    np.zeros(off_duration_short),
    np.ones(on_duration),
    np.zeros(off_duration_long)
])


def makeSignal(ref, sample_rate):
    # Define the carrier frequency
    carrier_frequency = 434e6

    # Generate the time vector for the base signal
    t = np.arange(len(ref)) / sample_rate

    # Generate the carrier signal
    carrier_signal = np.cos(2 * np.pi * carrier_frequency * t)

    # Modulate the base signal with the carrier
    modulated_signal = ref * carrier_signal

    return modulated_signal


def addTimeShift(s, t0, sample_rate):
    n = len(s)
    fft_signal = scipy.fft.fft(s)
```

```
    # Calculate the frequencies for each component
    freqs = np.fft.fftfreq(n, d=1 / sample_rate)

    # Create the phase shift array for the given time shift t0
    phase_shifts = np.exp(-1j * 2 * np.pi * freqs * t0)

    # Apply the phase shifts to the FFT signal
    shifted_fft_signal = fft_signal * phase_shifts

    # Compute the inverse FFT to get the time-shifted signal
    shifted_signal = scipy.fft.ifft(shifted_fft_signal)

    return shifted_signal


def zero_pad_array(arr, desired_length):
    """
    Function that adds additional zeros to the array
    :param arr: Array to be altered
    :param desired_length: Length which the altered array needs to be
    :return: The altered array
    """
    current_length = len(arr)
    if current_length < desired_length:
        padding = desired_length - current_length
        arr = np.pad(arr, (0, padding), 'constant', constant_values=(0,))
    return arr


def makePeak(ref, samples1):
    """
    Function to calculate the time difference and phase difference between the
    ↪  received signal and the reference signal
    :param ref: Reference signal
    :param samples1: Received signal
    :return: The time delay and phase difference
    """
    downsampling_factor = 1
    ref = ref[::downsampling_factor]
    samples1 = samples1[::downsampling_factor]

    length_ref = len(ref)
    length_samples = len(samples1)

    total_length = length_ref + length_samples - 1

    ref = zero_pad_array(ref, total_length)
    samples1 = zero_pad_array(samples1, total_length)

    mult1 = np.flip(scipy.fft.fft(np.conj(ref))) * scipy.fft.fft(samples1)
    ifft = np.abs(scipy.fft.ifft(mult1))
```

```
    td = np.linspace(0, total_length / sample_rate, total_length)
    # print(td)
    est_delay = td[np.argmax(np.abs(ifft))] * downsampling_factor

    T = 1
    phase_delay = 2 * np.pi * 1/T * est_delay
    phase_delay_degrees = np.rad2deg(phase_delay)

    return est_delay, phase_delay


def time_delay(ref, samples1, samples2):
    """
    Function to calculate the time difference and phase difference between two
    ↪  signal
    :param ref: Reference signal
    :param samples1: Received signal 1
    :param samples2: Received signal 2
    :return: The time difference and phase difference
    """
    downsampling_factor = 1
    ref = ref[::downsampling_factor]
    samples1 = samples1[::downsampling_factor]
    samples2 = samples2[::downsampling_factor]

    length_ref = len(ref)
    length_samples = len(samples1)

    total_length = length_ref + length_samples - 1

    ref = zero_pad_array(ref, total_length)
    samples1 = zero_pad_array(samples1, total_length)
    samples2 = zero_pad_array(samples2, total_length)

    mult1 = np.flip(scipy.fft.fft(np.conj(ref))) * scipy.fft.fft(samples1)
    mult2 = np.flip(scipy.fft.fft(np.conj(ref))) * scipy.fft.fft(samples2)

    if np.argmax(np.abs(scipy.fft.ifft(mult2))) >
    ↪  np.argmax(np.abs(scipy.fft.ifft(mult1))):
        ifft = np.abs(scipy.fft.ifft(mult2 / mult1))

    else:
        ifft = np.abs(scipy.fft.ifft(mult1 / mult2))

    td = np.linspace(0, total_length / sample_rate, total_length)
    est_delay = td[np.argmax(np.abs(ifft))] * downsampling_factor

    T = 1
    phase_delay = 2 * np.pi * 1/T * est_delay
    phase_delay_degrees = np.rad2deg(phase_delay)

    return est_delay, phase_delay
```

```
def makeNoise(signal):
    target_snr_db = 20

    x_watts = signal ** 2
    # Calculate signal power and convert to dB
    sig_avg_watts = np.mean(x_watts)
    sig_avg_db = 10 * np.log10(sig_avg_watts)
    # Calculate noise according to [2] then convert to watts
    noise_avg_db = sig_avg_db - target_snr_db
    noise_avg_watts = 10 ** (noise_avg_db / 10)
    # Generate a sample of white noise
    mean_noise = 0
    noise_volts = np.random.normal(mean_noise, np.sqrt(noise_avg_watts),
    ↪  len(x_watts))

    # Add the noise to the data.
    return signal + noise_volts


modulated_ref = makeSignal(ref, sample_rate)

shifted = addTimeShift(modulated_ref, 0.05, sample_rate)
shifted2 = addTimeShift(modulated_ref, 0.08, sample_rate)

shifted = makeNoise(shifted)
shifted2 = makeNoise(shifted2)

delay1, phase1 = makePeak(modulated_ref, shifted)
print(F"The time difference between the reference and the shifted signal is:
↪  {delay1} seconds."
      F" The phase difference is: {np.rad2deg(phase1)} degrees.")

delay2, phase2 = makePeak(modulated_ref, shifted2)
print(F"The time difference between the reference and the shifted signal is:
↪  {delay2} seconds."
      F" The phase difference is: {np.rad2deg(phase2)} degrees.")

delay3, phase3 = time_delay(modulated_ref, shifted, shifted2)
print(F"The time difference between the two shifted signals is: {delay3} seconds."
      F" The phase difference is: {np.rad2deg(phase3)} degrees.")

plot(modulated_ref, shifted, shifted2, np.rad2deg(phase3))
```

C.6 Phase difference between the two dongles

```
import numpy as np
import scipy
import matplotlib.pyplot as plt
import Receiver
from matplotlib.animation import FuncAnimation
import time
```

```
center_freq = 434e6
sample_rate = 2.048e6
receiver = Receiver.Receiver(center_freq, sample_rate, device_index=0)
receiver2 = Receiver.Receiver(center_freq, sample_rate, device_index=1)
receiver.start_receiving()
receiver2.start_receiving()
duration = 5  # Duration in seconds
wavelength = 3e8 / center_freq  # Speed of light divided by center frequency


on_duration = int(0.03 * sample_rate)
off_duration_short = int(0.03 * sample_rate)
off_duration_long = int(0.91 * sample_rate)

# Make the reference signal
ref = np.concatenate([
    np.ones(on_duration),
    np.zeros(off_duration_short),
    np.ones(on_duration),
    np.zeros(off_duration_long)
])


def makeSignal(ref, sample_rate):
    # Define the carrier frequency
    carrier_frequency = 434e6

    # Generate the time vector for the base signal
    t = np.arange(len(ref)) / sample_rate

    # Generate the carrier signal
    carrier_signal = np.cos(2 * np.pi * carrier_frequency * t)

    # Modulate the base signal with the carrier
    modulated_signal = ref * carrier_signal

    return modulated_signal


def zero_pad_array(arr, desired_length):
    """
    Function that adds additional zeros to the array
    :param arr: Array to be altered
    :param desired_length: Length which the altered array needs to be
    :return: The altered array
    """
    current_length = len(arr)
    if current_length < desired_length:
        padding = desired_length - current_length
        arr = np.pad(arr, (0, padding), 'constant', constant_values=(0,))
    return arr
```

```
def makePeak(ref, samples1):
    """
    Function to calculate the time difference and phase difference between the
    ↪  received signal and the reference signal
    :param ref: Reference signal
    :param samples1: Received signal
    :return: The time delay and phase difference
    """
    begin_time = time.time()

    downsampling_factor = 1
    ref = ref[::downsampling_factor]
    samples1 = samples1[::downsampling_factor]

    length_ref = len(ref)
    length_samples = len(samples1)

    total_length = length_ref + length_samples - 1

    ref = zero_pad_array(ref, total_length)
    samples1 = zero_pad_array(samples1, total_length)

    mult1 = np.flip(scipy.fft.fft(np.conj(ref))) * scipy.fft.fft(samples1)
    ifft = np.abs(scipy.fft.ifft(mult1))

    td = np.linspace(0, total_length / sample_rate, total_length)
    # print(td)
    est_delay = (td[np.argmax(np.abs(ifft))] * downsampling_factor) % 1

    T = 1
    phase_delay = 2 * np.pi * 1 / T * est_delay
    phase_delay_degrees = np.rad2deg(phase_delay)

    time_delta = time.time() - begin_time
    print(F"Elapsed time  is {time_delta} seconds")

    return est_delay, phase_delay

def time_delay(ref, samples1, samples2):
    """
    Function to calculate the time difference and phase difference between two
    ↪  signal
    :param ref: Reference signal
    :param samples1: Received signal 1
    :param samples2: Received signal 2
    :return: The time difference and phase difference
    """
    start_time = time.time()

    downsampling_factor = 1
    ref = ref[::downsampling_factor]
    samples1 = samples1[::downsampling_factor]
```

```
    samples2 = samples2[::downsampling_factor]

    length_ref = len(ref)
    length_samples = len(samples1)

    total_length = length_ref + length_samples - 1

    ref = zero_pad_array(ref, total_length)
    samples1 = zero_pad_array(samples1, total_length)
    samples2 = zero_pad_array(samples2, total_length)

    mult1 = np.flip(scipy.fft.fft(np.conj(ref))) * scipy.fft.fft(samples1)
    mult2 = np.flip(scipy.fft.fft(np.conj(ref))) * scipy.fft.fft(samples2)


    if np.argmax(np.abs(scipy.fft.ifft(mult2))) >
    ↪  np.argmax(np.abs(scipy.fft.ifft(mult1))):
        ifft = np.abs(scipy.fft.ifft(mult2 / mult1))

    else:
        ifft = np.abs(scipy.fft.ifft(mult1 / mult2))

    td = np.linspace(0, total_length / sample_rate, total_length)
    est_delay = (td[np.argmax(np.abs(ifft))] * downsampling_factor) % 1

    # plot(ref, samples1, mult1)

    T = 1
    phase_delay = 2 * np.pi * 1/T * est_delay
    phase_delay_degrees = np.rad2deg(phase_delay)

    delta_time = time.time() - start_time
    print(F"Elapsed time {delta_time} seconds. ")

    return est_delay, phase_delay


modulated_ref = makeSignal(ref, sample_rate)

# Create a figure and axis for plotting
fig, (ax1, ax2) = plt.subplots(2, 1, figsize=(10, 8))

x_data = np.linspace(-duration, 0, int(duration * sample_rate))
y_data1 = np.zeros(int(duration * sample_rate))
phase_data = []

line1, = ax1.plot(x_data, y_data1, lw=2, label='Receiver 1')
line2, = ax2.plot([], [], lw=2, label='Phase Difference')

ax1.set_ylim(0, 2)
ax1.set_xlabel('Time (s)')
ax1.set_ylabel('Amplitude')
ax1.set_title('Received Signal - RTL-SDR 1')
```

```
ax1.legend(loc='upper right')

ax2.set_ylim(0, 360)
ax2.set_xlim(-duration, 0)
ax2.set_xlabel('Time (s)')
ax2.set_ylabel('Phase Difference (degrees)')
ax2.set_title('Phase Difference')
ax2.legend(loc='upper right')


def init():
    """
    Initialize the plot
    :return:
    """
    line1.set_ydata(np.zeros_like(x_data))
    line2.set_data([], [])
    return line1, line2


def update(frame):
    """
    Function that updates the plot real-time
    :return: The lines to plot
    """
    start_time = time.time()
    samples1 = receiver .get_samples()
    time_received_first = time.time()
    samples2 = receiver2.get_samples()
    time_received_second = time.time()

    print(F"The first samples are saved after {time_received_first - start_time}
    ↪  seconds. The second samples are saved "
          F"{time_received_second - time_received_first} seconds after that.")


    if samples1 is not None:
        # Update the plot data for the received signal
        y_data1[:] = np.abs(samples1[:int(duration * sample_rate)])
        line1.set_ydata(y_data1)

        # Calculate phase difference
        delay3, phase3 = time_delay(modulated_ref, samples1, samples2)
        print(F"The time difference between the two shifted signals is: {delay3}
        ↪  seconds."
              F" The phase difference is: {np.rad2deg(phase3)} degrees.")

        phase_difference_degrees = np.degrees(phase3)  # Convert radians to degrees

        # Update the phase difference plot
        phase_data.append(phase_difference_degrees)
        if len(phase_data) > len(x_data):
            phase_data.pop(0)
```

```
        line2.set_data(np.linspace(-duration, 0, len(phase_data)), phase_data)

    return line1, line2


# Set up the animation
ani = FuncAnimation(fig, update, init_func=init, blit=True, interval=50,
↪  cache_frame_data=False)

# Show the plot
plt.tight_layout()
plt.show()

receiver.stop_receiving()
receiver2.stop_receiving()
```

C.7 Received signal strength indication measurement

```
import numpy as np
import scipy
import matplotlib.pyplot as plt
from matplotlib.animation import FuncAnimation
import Receiver  # Ensure this module is correctly implemented and imported

center_freq = 434e6
sample_rate = 2.048e6
duration = 5  # Duration in seconds

on_duration = int(0.03 * sample_rate)
off_duration_short = int(0.03 * sample_rate)
off_duration_long = int(0.91 * sample_rate)


receiver = Receiver.Receiver(center_freq, sample_rate, device_index=0)
receiver.start_receiving()

# Create a figure and axis for plotting
fig, (ax1, ax2) = plt.subplots(2, 1, figsize=(10, 8))

x_data = np.linspace(-duration, 0, int(duration * sample_rate))
y_data1 = np.zeros(int(duration * sample_rate))
y_data2 = np.zeros(int(duration * sample_rate))

line1, = ax1.plot(x_data, y_data1, lw=2, label='Receiver 1')
line2, = ax2.plot([], [], lw=2, label='Received signal strength indication')

ax1.set_ylim(0, 2)
ax1.set_xlabel('Time (s)')
ax1.set_ylabel('Amplitude')
ax1.set_title('Received Signal - RTL-SDR 1')
ax1.legend(loc='upper right')

ax2.set_ylim(-100, 20)
```

```
ax2.set_xlim(-duration, 0)
ax2.set_xlabel('Time (s)')
ax2.set_ylabel('RSSI (dB)')
ax2.set_title('RSSI')
ax2.legend(loc='upper right')

text = ax2.text(0.5, 0.5, '')
RSSI_data = []
RSSI_data_not_saved = []

def init():
    """
    Initialize the plot
    :return:
    """
    line1.set_ydata(np.zeros_like(x_data))
    line2.set_ydata(np.zeros_like(x_data))
    return line1, line2

def update(frame):
    """
    Function that updates the plot real-time
    :return: The lines to plot
    """
    samples1 = receiver.get_samples()

    if samples1 is not None:
        # Update the plot data for the received signal
        y_data1[:] = np.abs(samples1[:int(duration * sample_rate)])
        line1.set_ydata(y_data1)
        # print(F"Max value {np.max(samples1)}")

        # Calculate RSSI
        power = np.abs(np.max(samples1)) ** 2
        print(power)

        RSSI = 10 * np.log10(power + 1e-12)

        # Update the RSSI data
        y_data2[:-1] = y_data2[1:]
        y_data2[-1] = RSSI
        line2.set_ydata(y_data2)

        RSSI_data.append(RSSI)
        RSSI_data_not_saved.append(RSSI)

        if len(RSSI_data_not_saved) > len(x_data):
            RSSI_data_not_saved.pop(0)

        line2.set_data(np.linspace(-duration, 0, len(RSSI_data_not_saved)),
        ↪  RSSI_data_not_saved)

        text.set_text(f'{RSSI:.03f}')
```

```
        text.set_position((-2, -40))

    return line1, line2, text,

# Set up the animation
ani = FuncAnimation(fig, update, init_func=init, blit=True, interval=10,
↪  cache_frame_data=False)

# Show the plot
plt.tight_layout()
plt.show()

# Stop receiving samples when closing the plot
receiver.stop_receiving()

file = open("RSSI.txt", "w")
file.write(str(RSSI_data))
file.close()
```

C.8 Direct phase difference

D Calculations

D.1 Cramer Rao Bound calculation

The CRB is calculated using the signal power of -164.45 dB, a noise power of -134.96 dB, angular frequency ω_c ($2.7 \cdot 10^9$ rad/s, bandwidth B (1.5 MHz) and the measurement time T (0.128 s). The center frequency in radians is calculated by multiplying the center frequency with 2π.
The CRB 41 and Fisher 42 formula's are denoted again here.

$$CRB = J_{CR}^{-1} \tag{41}$$

$$J_{CR}(\tau) = 2\omega_c^2 \cdot \frac{TB}{2\pi}[1 + \frac{1}{12}\left(\frac{B}{\omega_c}\right)^2] \cdot SNR, \quad for \quad T >> \frac{2\pi}{B} \tag{42}$$

The SNR in this equation is calculated using

$$SNR = \frac{\frac{S}{N_0}^2}{1 + 2\frac{S}{N_0}} \tag{43}$$

Resulting in $SNR = 1.265 \cdot 10^{-6}$.
The Fisher equation is filled out as follows.

$$J_{CR} = 2 \cdot (2.7 \cdot 10^9)^2 \cdot \frac{0.128 \cdot 1.5 \cdot 10^6 \cdot 2\pi}{2\pi} \cdot [1 + \frac{1}{12} \cdot \left(\frac{1.5 \cdot 10^6 \cdot 2\pi}{2.7 \cdot 10^9}\right)^2] 1.265 \cdot 10^{-6} \tag{44}$$

This is equal to $3.603 \cdot 10^{18}$. Which gives a CRB of $4.620 \cdot 10^{-19}$. As the CRB gives variance of the unbiased estimator, taking the square root of the CRB will give the accuracy in seconds, $\Delta t = \sqrt{4.620 \cdot 10^{-19}} = 5.268 \cdot 10^{-10}$. Using this time difference the minimum angle of arrival can be calculated by rewriting Equation 45 into Equation 46 and using $d = \frac{\lambda}{2} = \frac{c}{2f}$

$$\Delta t = \frac{sin(\theta) \cdot d}{c} \tag{45}$$

$$\theta = sin^{-1}\left(\frac{\Delta t \cdot c}{d}\right) = sin^{-}1(\Delta t \cdot 2f_c) \tag{46}$$

This results in an angle of arrival of 0.5°.